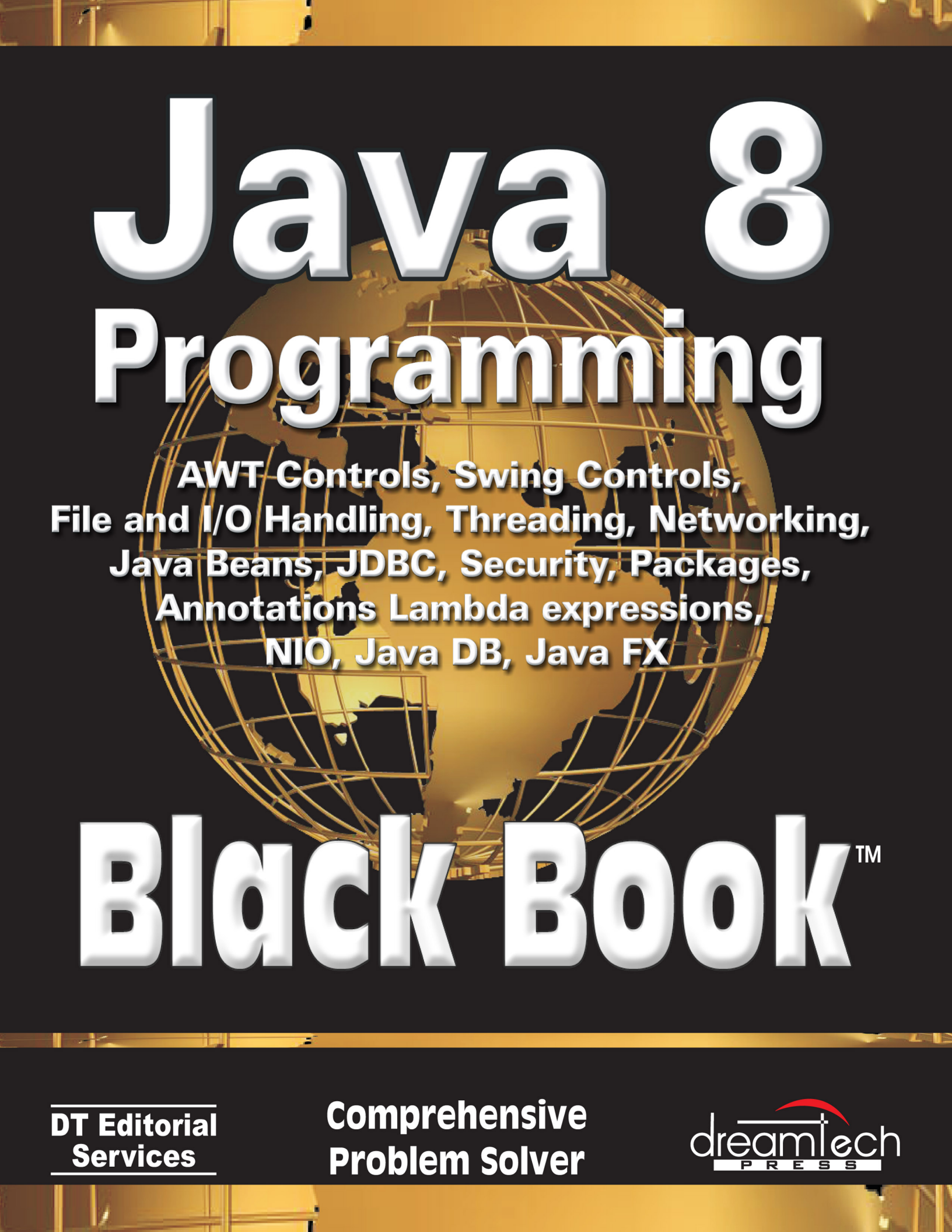
Java 8 Programming

AWT Controls, Swing Controls,
File and I/O Handling, Threading, Networking,
Java Beans, JDBC, Security, Packages,
Annotations Lambda expressions,
NIO, Java DB, Java FX



Black Book™

DT Editorial Services

Comprehensive Problem Solver

dreamtech PRESS



Java 8 Programming

AWT Controls, Swing Controls, File and I/O Handling, Threading, Networking, Java Beans, JDBC, Security, Packages, Annotations Lambda expressions, NIO, Java DB, Java FX

Black Book™

Authored by:

DT Editorial Services

Published by:

dreamtech PRESS

©Copyright 2015 by Dreamtech Press, 19-A, Ansari Road, Daryaganj, New Delhi-110002

Black Book is a trademark of Paraglyph Press Inc., 2246 E. Myrtle Avenue, Phoenix Arizona 85202, USA exclusively licensed in Indian, Asian and African continent to Dreamtech Press, India.

This book may not be duplicated in any way without the express written consent of the publisher, except in the form of brief excerpts or quotations for the purposes of review. The information contained herein is for the personal use of the reader and may not be incorporated in any commercial programs, other books, databases, or any kind of software without written consent of the publisher. Making copies of this book or any portion for any purpose other than your own is a violation of copyright laws.

Limits of Liability/disclaimer of Warranty: The author and publisher have used their best efforts in preparing this book. The author make no representation or warranties with respect to the accuracy or completeness of the contents of this book, and specifically disclaim any implied warranties of merchantability or fitness of any particular purpose. There are no warranties which extend beyond the descriptions contained in this paragraph. No warranty may be created or extended by sales representatives or written sales materials. The accuracy and completeness of the information provided herein and the opinions stated herein are not guaranteed or warranted to produce any particulars results, and the advice and strategies contained herein may not be suitable for every individual. Neither Dreamtech Press nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

Trademarks: All brand names and product names used in this book are trademarks, registered trademarks, or trade names of their respective holders. Dreamtech Press is not associated with any product or vendor mentioned in this book.

ISBN: 978-93-5119-758-4

ISBN: 978-93-5119-953-3 (ebk)

Edition: 2015

Printed at: Himal Impressions, Delhi



Contents at a Glance

Introduction xxxiii

Chapter 1: Essential Java 1

Chapter 2: Variables, Arrays, and Strings 31

Chapter 3: Operators, Conditionals, and Loops 75

Chapter 4: Object-Oriented Programming 105

Chapter 5: Inheritance, Inner Classes, and Interfaces 145

Chapter 6: AWT—Applets, Applications, and Event Handling 171

Chapter 7: AWT—Text Fields, Buttons, Checkboxes, Radio Buttons, and Layouts 215

Chapter 8: AWT—Lists, Choices, Text Areas, Scroll bars, and Scroll Panes 245

Chapter 9: AWT—Graphics, Images, Text, and Fonts 275

Chapter 10: AWT—Windows, Menus, and Dialog Boxes 315

Chapter 11: Swing—Applets, Applications, and Pluggable Look and Feel 351

Chapter 12: Swing—Text Fields, Buttons, Toggle Buttons, Checkboxes, and Radio Buttons 395

Chapter 13: Swing—Viewports, Scrolling, Sliders, Lists, Tables, and Trees 443

Chapter 14: Swing—Combo Boxes, Progress Bars, Tooltips, Separators, and Choosers 489

Chapter 15: Swing—Layered Panes, Tabbed Panes, Split Panes, and Layouts 521

Chapter 16: Swing—Menus and Toolbars 549

Chapter 17: Swing—Windows, Desktop Panes, Inner Frames, and Dialog Boxes 579

Chapter 18: Working with Streams, Files, and I/O Handling 615

Chapter 19: Working with Multiple Threads 673

Chapter 20: Networking and Security with Java 725

Chapter 21: Collections 767

Chapter 22: Creating Packages, Interfaces, JAR Files, and Annotations 823

Chapter 23: Working with Java Beans 845

Chapter 24: Introducing JDBC 875

Chapter 25: Images and Animation 927

Chapter 26: Java DB 951

Chapter 27: Java FX 969

Glossary 993

Index 997

Bonus chapters available with the book for Download

Chapter 1: Java and XML—Using the Document Object Model 1

Chapter 2: Java and XML—Using the Simple API for XML 31

Chapter 3: Understanding RMI 57

Chapter 4: Working with Servlets 3.1 83

Chapter 5: Dynamic Java Scripting 137

Chapter 6: Introduction to Groovy 159

Chapter 7: MXBeans and JMX 189

Chapter 8: Inclusion of Sound 213

Chapter 9: Filing and Printing Documents 251



Table of Contents

Introduction xxxiii

Chapter 1: Essential Java 1

In Depth 2

All about Java 2

Java Appears 3

All about Bytecodes 3

Features of Java 4

Java Environment 6

Java Development Kit 6

Java Platforms 6

Java Virtual Machine 8

Java API 8

Java Programs 9

What are Application Servers (Tomcat, JRun, WebSphere, WebLogic, etc.) 9

Program Design 11

Performance 11

Maintainability 11

Extensibility 12

Availability 12

New Features of Java SE 8 12

Immediate Solutions 15

Getting and Installing Java 15

What about CLASSPATH? 15

Subfolders Created by JDK 16

Creating Source Files 16

Knowing Java's Reserved Words 17
Creating an Application 19
Line 1—**public class App** 19
Line 2—**public static void main(String[] args)** 19
Line 3—**System.out.println("Hello from Java!");** 20
Compiling Code 20
Compiling Code: Using Command-Line Options 21
Cross-Compilation Options 22
Compiling Code: Checking for Deprecated Methods 23
Running Code 23
Running Code: Using Command-Line Options 25
Commenting Your Code 26
Import Java Packages and Classes 27
Finding Java Classes with **CLASSPATH** 28
Summary 30

Chapter 2: Variables, Arrays, and Strings 31

In Depth **..... 33**
Variables 33
Data Typing 34
Arrays 34
Strings 36

Immediate Solutions **..... 37**
What Data Types are Available? 37
Creating Integer Literals 37
Creating Floating-Point Literals 38
Creating Boolean Literals 39
Creating Character Literals 39
Creating String Literals 40
Creating Binary Literals 40
Using Underscores in Numeric Literals 41
Declaring Integer Variables 42
Declaring Floating-Point Variables 42
Declaring Character Variables 43
Declaring Boolean Variables 43
Initializing Variables Dynamically 44

Conversion between Data Types ... 45
Automatic Conversions ... 45
Casting to New Data Types ... 46
Declaring One-Dimensional Arrays ... 47
Creating One-Dimensional Arrays ... 47
Initializing One-Dimensional Arrays ... 48
Declaring Multi-Dimensional Arrays ... 48
Creating Multi-Dimensional Arrays ... 48
Initializing Multi-Dimensional Arrays ... 49
Creating Irregular Multi-Dimensional Arrays ... 50
Getting an the Length of an Array ... 51
Understanding General Form of Static Import ... 51
Importing Static Members ... 51
The **String** Class ... 52
Creating Strings ... 56
Getting String Length ... 57
Concatenating Strings ... 58
Getting Characters and Substrings ... 58
Searching For and Replacing Strings ... 59
Changing Case in Strings ... 60
Checking for Empty String ... 60
Formatting Numbers in Strings ... 60
The **StringBuffer** Class ... 61
Creating StringBuffers ... 63
Getting and Setting StringBuffer Lengths and Capacities ... 64
Setting Characters in String Buffers ... 64
Appending and Inserting Using StringBuffers ... 65
Deleting Text in StringBuffers ... 65
Replacing Text in StringBuffer ... 65
Using the **Wrapper** Class ... 66
Autoboxing and Unboxing of Primitive Types ... 67
Learning the Fundamentals of Varargs Methods ... 69
Overloading Varargs Methods ... 71
Learning the Ambiguity in Varargs Methods ... 72
Using Non-Reifiable Formal Parameters ... 73
Summary ... 74

Chapter 3: Operators, Conditionals, and Loops 75

In Depth **..... 77**

Operators 77
Conditionals 78
Loops 79

Immediate Solutions **..... 81**

Operator Precedence 81
Incrementing and Decrementing (**++** and **--**) 82
Unary **NOT** (**~** **And !**) 83
Multiplication and Division (***** and **/**) 83
Modulus (**%**) 83
Addition and Subtraction (**+** and **–**) 84
Shift Operators (**>>**, **>>>**, and **<<**) 84
Relational Operators (**>**, **>=**, **<**, **<=**, **==**, and **!=**) 85
Bitwise and Bitwise Logical **AND, XOR,** and **OR** (**&, ^**, and **|**) 85
Logical (**&&** and **||**) 87
The **if-then-else** Operator (**? :**) 87
Assignment Operators (**=** and **[operator]=**) 88
Using the **Math** Class 89
Changes in the **Math** Class 90
Class **StrictMath** 90
Comparing Strings 91
The **if** Statement 92
The **else** Statement 92
Nested **if** Statement 93
The **if-else** Ladders 93
The **switch** Statement 93
Using Strings in switch Statement 95
The **while** Loop 95
The **do-while** Loop 96
The **for** Loop 97
The **for-each** Loop 99
Supporting **for-each** in Your Own Class 101
A (Poor) Solution 101
Significance of **for-each** 101
Nested Loops 102

Using the **break** Statement ... 103
Using the **continue** Statement ... 103
Summary ... 104

Chapter 4: Object-Oriented Programming ... 105

In Depth **... 107**

Classes ... 108
Objects ... 108
Data Members ... 108
Methods ... 108
Inheritance ... 109
Exception Handling ... 109
Debugging ... 110

Immediate Solutions **... 111**

Declaring and Defining Classes ... 111
Declaring and Creating Objects ... 112
Creating Instance Variables ... 114
Setting Variable Access ... 114
Creating Class Variables ... 115
Creating Methods ... 116
Setting Method Access ... 116
Passing Parameters to Methods ... 117
Command-Line Arguments Passed to **main() method** ... 118
Returning Values from Methods ... 118
Creating Class Methods ... 119
Creating Data Access Methods ... 120
Creating Constructors ... 120
Passing Parameters to Constructors ... 121
A Full Class Example ... 121
Understanding Variable Scope ... 122
Using Recursion ... 123
Garbage Collection and Memory Management ... 123
Avoiding Circular References ... 124
Garbage Collection and the **finalize()** Method ... 125
Overloading Methods ... 125
Overloading Constructors ... 126
Passing Objects to Methods ... 126

Passing Arrays to Methods ... 127
Using the **this** Keyword ... 128
Returning Objects from Methods ... 129
Returning Arrays from Methods ... 129
The ProcessBuilder and StringBuilder Classes ... 129
The **ProcessBuilder** Class ... 130
The **StringBuilder** Class ... 131
Catching an Exception ... 132
Nesting the **try** Statements ... 136
Using the **finally** Clause ... 137
Throwing Exceptions ... 138
Creating a Custom Exception ... 139
Catching Multiple Exception Types ... 139
Rethrowing Exceptions with Improved Type Checking ... 140
Debugging Java Programs ... 141
Summary ... 144

Chapter 5: Inheritance, Inner Classes, and Interfaces ... 145

In Depth **... 146**

Why Inheritance? ... 146
Why Interfaces? ... 147
Why Inner Classes? ... 147
Lambda Expressions ... 148
Method References ... 148

Immediate Solutions **... 150**

Creating a Subclass ... 150
Access Specifiers and Inheritance ... 150
Calling Superclass Constructors ... 151
Creating Multilevel Inheritance ... 153
Handling Multilevel Constructors ... 154
Overriding Methods ... 155
Accessing Overridden Members ... 155
Using Superclass Variables with Subclassed Objects ... 156
Dynamic Method Dispatch (Runtime Polymorphism) ... 157
Creating Abstract Classes ... 158
Stopping Overriding with **final** ... 159
Stopping Inheritance with **final** ... 159

Creating Constants with **final** ... 160
Is-a vs. Has-a Relationships ... 160
The Java **Object** Class ... 161
Using Interfaces for Multiple Inheritance ... 162
The **Readable** Interface ... 163
The **Appendable** Interface ... 164
The **Iterable** Interface ... 164
Creating **Iterable** Objects ... 166
Creating Inner Classes ... 168
Creating Anonymous Inner Classes ... 168
Using Lambda Expression ... 169
Default Methods ... 169
Summary ... 170

Chapter 6: AWT—Applets, Applications, and Event Handling ... 171

In Depth **... 172**

The Abstract Windowing Toolkit ... 172
Applets ... 173
Applications ... 174
Handling Events ... 174

Immediate Solutions **... 175**

Using the Abstract Window Toolkit ... 175
Creating Applets ... 186
Using the **<APPLET>** HTML Tag ... 188
Handling Non-Java Browsers ... 189
Embedding **<APPLET>** Tags in Code ... 189
Using the init, start, stop, destroy, paint, and update Methods ... 190
Drawing Graphics in Applets ... 191
Reading Parameters in Applets ... 191
Using Java Consoles in Browsers ... 191
Adding Controls to Applets: Text Fields ... 192
Adding Controls to Applets: Buttons ... 193
Handling Events ... 194
Standard Event Handling ... 195
Using Delegated Classes ... 197
Using Action Commands ... 199
Handling Events the Old Way ... 199

Extending Components ... 200
Using Adapter Classes ... 200
Using Anonymous Inner Adapter Classes ... 202
Creating Windowed Applications ... 202
Exiting an Application When Its Window Is Closed ... 206
Applications You Can Run as Applets ... 206
Setting Applet Security Policies ... 207
Other Facilities in the **java.awt** Package ... 209
System Tray ... 209
Splash Screen ... 211
Dialog Modality ... 213
Gif Writer ... 213
Text Antialiasing ... 214
Summary ... 214

Chapter 7: AWT—Text Fields, Buttons, Checkboxes, Radio Buttons, and Layouts ... 215

In Depth **... 216**

Text Fields ... 216
Buttons ... 216
Checkboxes ... 216
Radio Buttons ... 216
Layouts ... 216

Immediate Solutions **... 218**

Using Text Fields ... 218
Using Labels ... 220
Using Buttons ... 221
Using Checkboxes ... 224
Using Radio Buttons ... 227
Layout Managers ... 228
Flow Layouts ... 228
Grid Layouts ... 231
Using Panels ... 232
Border Layouts ... 234
Card Layouts ... 236
Grid Bag Layouts ... 238
Using Insets and Padding ... 243

Creating Your Own Layout Manager ... 244
Summary ... 244

Chapter 8: AWT—Lists, Choices, Text Areas, Scroll bars, and Scroll Panes ... 245

In Depth **... 246**

Lists ... 246
Choices ... 246
Text Areas ... 246
Scroll bars ... 246
Scroll Panes ... 247

Immediate Solutions **... 248**

Using Text Areas ... 248
Replacing Text in Text Areas ... 251
Searching and Selecting Text in Text Areas ... 252
Using Lists ... 253
Using Multiple-Selection Lists ... 257
Using Choice Controls ... 260
Using Scroll bars ... 265
Scroll bars and Border Layouts ... 269
Using Scroll Panes ... 272
Summary ... 274

Chapter 9: AWT—Graphics, Images, Text, and Fonts ... 275

In Depth **... 276**

Graphics ... 276
Images ... 276
Text and Fonts ... 276
The Keyboard and Mouse ... 276

Immediate Solutions **... 277**

Using the Mouse ... 277
Using the Keyboard ... 279
Using Fonts ... 283
Using Images ... 289
Resizing Images ... 291
Drawing Graphics ... 292
Drawing Lines ... 297

Drawing Ovals ... 298
Drawing Rectangles ... 298
Drawing Rounded Rectangles ... 299
Drawing Freehand ... 299
Drawing Arcs ... 299
Drawing Polygons ... 299
Setting Drawing Modes ... 300
Selecting Colors ... 300
Using Canvases ... 303
Using the **ImageObserver** Interface ... 305
Using the **MediaTracker** Class ... 306
Working Pixel by Pixel: The **PixelGrabber** and **MemoryImageSource** Classes ... 308
Brightening Images ... 311
Converting Images to Grayscale ... 312
Embossing Images ... 313
Summary ... 314

Chapter 10: AWT—Windows, Menus, and Dialog Boxes ... 315

In Depth **... 316**

Windows ... 316
Menus ... 316
Dialog Boxes ... 316

Immediate Solutions **... 318**

Creating Frame Windows ... 318
Showing and Hiding Windows ... 319
Handling Window Events ... 320
Automatically Hiding Windows upon Closing ... 323
Using the **Window** Class ... 323
Creating Menus ... 330
Creating a **MenuBar** Object ... 331
Creating **Menu** Objects ... 332
Creating **MenuItem** Objects ... 333
Handling Menu Events ... 335
More Menu Options ... 336
Adding Menu Separators ... 338
Disabling Menu Items ... 338

Adding Checkboxes to Menus ... 339
Creating Submenus ... 341
Pop-Up Menus ... 342
Dialog Boxes ... 344
File Dialog Boxes ... 348
Summary ... 350

Chapter 11: Swing—Applets, Applications, and Pluggable Look and Feel ... 351

In Depth **... 352**

The Java Foundation Classes ... 352
Swing ... 353
Heavyweight versus Lightweight Components ... 354
Swing Features ... 355
Graphics Programming Using Panes ... 356
Model View Controller Architecture ... 356

Immediate Solutions **... 358**

Working with Swing ... 358
Preparing to Create a Swing Applet ... 364
Understanding Root Panes ... 366
Understanding Layered Panes ... 368
Understanding Content Panes ... 370
Working with Content Panes ... 370
Creating a Swing Applet ... 372
Painting in Swing vs. AWT ... 372
Displaying Controls in Swing Vs AWT ... 372
Using the **JPanel** Class ... 373
Creating a Swing Application ... 374
Closing **JFrame** Windows ... 377
Using Insets for Selecting Component Borders ... 379
Using Insets ... 381
The Synth Skinnable Look and Feel ... 383
The Pluggable Look and Feel ... 386
The Pluggable Look and Feel for Components ... 390
The Nimbus Look and Feel ... 392
Summary ... 394

Chapter 12: Swing—Text Fields, Buttons, Toggle Buttons, Checkboxes, and Radio Buttons 395

In Depth **..... 397**

Labels and Text Fields 397
Password Field 397
Text Area 397
Editor Pane 397
Text Pane 398
Text Component Printing 398
Buttons 398
Toggle Buttons 398
Checkboxes and Radio Buttons 398

Immediate Solutions **..... 399**

Using Labels 399
Using Image Icons 401
Using Images in Labels 402
Using Text Fields 403
Setting Text Field Alignment 405
Creating Password Field 406
Creating Text Areas 407
Customizing a Text Area 409
Creating Editor Panes 409
Using HTML in Editor Panes 411
Using RTF Files in Editor Panes 412
Creating Text Panes 412
Inserting Images and Controls into Text Panes 412
Setting Text Pane Text Attributes 414
Working with Sound in Applets 416
Working with Sound in Application 417
Abstract Button: The Foundation of Swing Buttons 417
Using Buttons 421
Displaying Images in Buttons 424
Using Rollover and Disabled Images 426
Default Buttons and Mnemonics 427
Using Toggle Buttons 430
Creating Toggle Button Groups 432

Using Checkboxes ... 433
Using Radio Buttons ... 436
Using Checkbox and Radio Button Images ... 440
Getting and Setting the State of Checkboxes and Radio Buttons ... 441
Summary ... 442

Chapter 13: Swing—Viewports, Scrolling, Sliders, Lists, Tables, and Trees ... 443

In Depth **... 445**

Viewports ... 445
Scroll Panes ... 445
Sliders ... 445
Scrollbars ... 445
Lists ... 445
Tables ... 445
Trees ... 446

Immediate Solutions **... 447**

Handling Viewports ... 447
Creating Scroll Panes ... 450
Creating Scroll Pane Headers and Borders ... 454
Scrolling Images ... 455
Creating Sliders ... 456
Filling a Slider ... 459
Painting Slider Tick Marks ... 460
Painting Slider Labels ... 460
Setting the Slider Extent ... 461
Creating Scrollbars ... 462
Creating Lists ... 465
Handling Multiple List Selections ... 469
List Selection Modes ... 469
Displaying Images in Lists ... 470
Creating a Custom List Model ... 471
Creating a Custom List Cell Renderer ... 472
Handling Double Clicks in Lists ... 472
Drag and Drop Support ... 473
Creating a Table ... 476
Table Sorting and Filtering ... 478
Sorting Rows ... 478

Filtering Table Rows ... 480
Adding Rows and Columns to Tables at Runtime ... 482
Creating Trees ... 483
Adding Data to Trees ... 484
Handling Tree Events ... 486
Summary ... 488

Chapter 14: Swing—Combo Boxes, Progress Bars, Tooltips, Separators, and Choosers ... 489

In Depth **... 490**

Combo Boxes ... 490
Progress Bars ... 490
Choosers ... 490
Tooltips ... 490
Separators ... 491

Immediate Solutions **... 492**

Creating Combo Boxes ... 492
Handling Combo Box Selection Events ... 496
Creating Editable Combo Boxes ... 497
Adding Images to Combo Boxes ... 499
Creating a Combo Box Model ... 500
Creating a Combo Box Custom Renderer ... 500
Creating Progress Bars ... 500
Updating Progress Bars ... 503
Handling Progress Bar Events ... 504
Creating Tooltips ... 505
Creating Separators ... 507
Resizing Separators Automatically ... 508
Creating Color Choosers ... 510
Creating File Choosers ... 512
Creating File Chooser Filters ... 518
Summary ... 520

Chapter 15: Swing—Layered Panes, Tabbed Panes, Split Panes, and Layouts ... 521

In Depth **... 522**

Layered Panes ... 522
Tabbed Panes ... 522

Split Panes ... 522
Layouts ... 522
Immediate Solutions ... **523**
Understanding Swing Components and Z-order ... 523
Making Swing Components Transparent ... 524
Using Layered Panes ... 525
Creating Tabbed Panes ... 527
Specifying Tab Placement in the Tabbed Panes ... 532
Using Split Panes ... 535
Making Split Panes One-Touch Expandable ... 538
Setting Split Pane Orientation ... 539
Setting Split Pane Divider Size ... 540
Using the Box Layout Manager ... 541
Using the **Box** Class ... 543
Using the Overlay Layout Manager ... 547
Summary ... 548

Chapter 16: Swing—Menus and Toolbars ... 549

In Depth ... **550**
Menus ... 550
Toolbars ... 550
Immediate Solutions ... **551**
Creating a Menu Bar ... 551
Creating a Menu ... 552
Creating a Menu Item ... 555
Creating a Basic Menu System ... 557
Adding Images to Menu Items ... 559
Creating Checkbox Menu Items ... 560
Creating Radio Button Menu Items ... 562
Creating Submenus ... 564
Creating Menu Accelerators ... 565
Enabling/Disabling and Changing Menu Items at Runtime ... 567
Adding and Removing Menu Items at Runtime ... 568
Adding Buttons and Other Controls to Menus ... 569
Creating Pop-Up Menus ... 570
Creating Toolbars ... 574

Adding Combo Boxes and Other Controls to Toolbars ... 576
Summary ... 578

Chapter 17: Swing—Windows, Desktop Panes, Inner Frames, and Dialog Boxes ... 579

In Depth **... 580**

Windows ... 580
Dialog Boxes ... 580

Immediate Solutions **... 581**

Creating a Window ... 581
Designing Shaped and Translucent Windows ... 583
Creating Translucent Windows ... 583
Creating Different Shaped Windows ... 587
Creating a Frame Window ... 588
Creating a Desktop Pane ... 589
Creating Internal Frames ... 591
Using JOptionPane to Create Dialog Boxes ... 598
Creating Option Pane Confirmation Dialog Boxes ... 604
Creating Option Pane Message Dialog Boxes ... 605
Creating Option Pane Text Field Input Dialog Boxes ... 606
Creating Option Pane Combo Box Input Dialog Boxes ... 608
Creating Option Pane Internal Frame Dialog Boxes ... 609
Creating Dialog Boxes with JDialog ... 609
Getting Input from Dialog Boxes Created with JDialog ... 613
Summary ... 614

Chapter 18: Working with Streams, Files, and I/O Handling ... 615

In Depth **... 616**

Streams, Readers, and Writers ... 616
NIO ... 616
Essentials in NIO ... 616
Buffers ... 617
Charsets and Selectors ... 620
Enhancements in NIO with Java 8 ... 620
The Path Interface ... 620
The **Files** Class ... 622
The **Paths** Class ... 624
The File Attribute Interfaces ... 625

The **FileSystem** Class ... 626
The **FileSystems** Class ... 627
The **FileStore** Class ... 628
Prospects of NIO ... 629
Immediate Solutions ... **630**
Working with Streams ... 630
The **InputStream** Class ... 630
The **OutputStream** Class ... 631
The **ByteArrayInputStream** Class ... 631
The **ByteArrayOutputStream** Class ... 633
The **BufferedInputStream** Class ... 635
The **BufferedOutputStream** Class ... 636
The **FileInputStream** Class ... 637
The **FileOutputStream** Class ... 638
Working with the **Reader** Class ... 640
Working with the **Writer** Class ... 641
Accepting Input from the Keyboard with the **InputStreamReader** Class ... 641
Working with the **OutputStreamWriter** Class ... 643
Working with Files ... 643
Using the **File** Class ... 643
Using the **FileReader** Class ... 647
Using the **FileWriter** Class ... 648
Working with the **RandomAccessFile** Class ... 649
Working with Character Arrays ... 651
Using the **CharArrayReader** Class ... 651
Using the **CharArrayWriter** Class ... 653
Working with Buffers ... 654
Using the **BufferedReader** Class ... 654
Using the **BufferedWriter** Class ... 656
Working with the **PushbackReader** Class ... 656
Working with the **PrintWriter** Class ... 657
Working with the **StreamTokenizer** Class ... 659
Implementing the **Serializable** Interface ... 661
Working with the **Console** Class ... 665
Working with the Clipboard ... 667
Working with the Printer ... 667

Printing with the **Formatter** Class ... 668
Using the **System.out.printf()** Method ... 668
Using the **String.format()** Method ... 669
Formatting Dates Using the **String.format()** Method ... 669
Using the **java.util.Formatter** Class ... 670
Scanning Input with the **Scanner** class ... 671
Summary ... 672

Chapter 19: Working with Multiple Threads ... 673

In Depth **... 674**

Using Threads in Java ... 674
Life Cycle of a Thread ... 675
Synchronization of Threads ... 676
Multithreaded Custom Class Loader ... 677

Immediate Solutions **... 678**

Getting the Main Thread ... 678
Naming a Thread ... 678
Pausing a Thread ... 679
Creating a Thread with the **Runnable** Interface ... 681
Creating a Thread with the **Thread** Class ... 684
Creating Multiple Threads ... 686
Joining Threads ... 688
Checking if a **Thread** Is Alive ... 689
Setting Thread Priority and Stopping Threads ... 690
Synchronizing ... 692
Synchronizing Code Blocks ... 693
Synchronizing Methods ... 694
Synchronizing Classes ... 696
Communicating between Threads ... 704
Suspending and Resuming Threads ... 705
Creating Graphics Animation with Threads ... 707
Eliminating Flicker in Graphics Animation Created Using Threads ... 710
Suspending and Resuming Graphics Animation ... 711
Using Double Buffering ... 713
Simplifying Producer-Consumer with the **Queue** Interface ... 715
Implementing Concurrent Programming ... 717
Limitations of Synchronization Previous to J2SE 5 ... 718

Simplifying Servers Using the Concurrency Utilities 718
Knowing Various Concurrency Utilities 719
Learning about the **java.util.concurrent** Package 719
Learning about the **java.util.concurrent.locks** Package 721
Reader/Writer Locks 721
Learning about the **java.util.concurrent.atomic** Package 722
Summary 724

Chapter 20: Networking and Security with Java 725

***In Depth* 726**

Basics of Networking 726
Networking Enhancements in Java SE 8 727
JDK 8 Security Enhancements 727
Sockets in Java 728
Client-Server Networking 729
Proxy Servers 729
Internet Addressing 730
Domain Name Service 730
Inet4Addresses and Inet6Addresses 730
The **URL** Class 731
The **URI** Class 731
URI Syntax and Components 732
TCP/IP and Datagram 732
Blackboard Assignment Retrieval Transaction 732

***Immediate Solutions* 734**

Understanding Networking Interfaces and Classes in the java.net Package 734
The Networking Interfaces and Classes 734
Understanding the **InetAddresses** 736
IP Addresses Scope 736
Host Name Resolution 736
Caching **InetAddress** 736
Factory() Method 739
Instance() Method 740
Creating and Using Sockets 740
Creating TCP Clients and Servers 743
TCP/IP Client Sockets 744

Understanding the **Whois** Example ... 746
TCP/IP Server Sockets ... 746
Submitting an HTML Form from a Java Program ... 748
Handling **URL** ... 749
Using the **URLConnection** Objects ... 752
Working with Datagrams ... 756
DatagramPacket ... 756
Datagrams Server and Client ... 758
Working with **BART** ... 759
Learning about the **java.security** Package ... 761
The Permission Class ... 764
The **Policy** Class ... 764
Summary ... 766

Chapter 21: Collections ... 767

In Depth **... 768**

The Collection Interfaces ... 768
The Collection Classes ... 768
The **Map** Interfaces ... 769
The **Map** Classes ... 769
Collections Framework Enhancements in Java SE 8 ... 769
Support for Lambda Expressions, Streams, and Aggregate Operations ... 770
Performance Improvement for HashMaps with Key Collisions ... 774
Improved Type Inference ... 774

Immediate Solutions **... 776**

Using the **Collection** Interface ... 776
The **Queue** Interface ... 777
The **List** Interface ... 777
The **Set** Interface ... 778
The **SortedSet** Interface ... 779
Using the **Collection** Classes ... 779
The **AbstractCollection** Class ... 779
The **AbstractList** Class ... 780
The **AbstractSequentialList** Class ... 781
The **ArrayList** Class ... 782
The **ArrayDeque** Class ... 784
The **LinkedList** Class ... 786

The **Generic** Class ... 788
The **HashSet** Class ... 789
The **TreeSet** Class ... 790
Using the **Comparator** Interface ... 792
Using the **Iterator** Interface ... 793
Using the **ListIterator** Interface ... 793
Using the **AbstractMap** Class ... 795
Using the **HashMap** Class ... 796
Using the **TreeMap** Class ... 799
Using the **Arrays** Class ... 801
Learning the Fundamentals of Enumerations ... 804
The **values()** and **valueOf()** Methods ... 806
Java Enumeration as a Class Type ... 807
Enumeration Inheriting Enum ... 808
The **Enumeration** Interface ... 810
The **Legacy** Classes and Interfaces ... 810
The **Vector** Class ... 811
The **Stack** Class ... 813
The **Dictionary** Class ... 814
The **Hashtable** Class ... 815
The **Properties** Class ... 817
Using the Aggregate Operations ... 818
Using the **java.util.function** Package ... 819
Summary ... 822

Chapter 22: Creating Packages, Interfaces, JAR Files, and Annotations ... 823

In Depth **... 824**

Packages and Interfaces ... 824
JAR Files ... 824
The Core Java API Package ... 824
The java.lang Package ... 826
The **java.lang.annotation** Subpackage ... 826
The **java.lang.instrument** Subpackage ... 826
The **java.lang.management** Subpackage ... 826
The **java.lang.reflect** Subpackage ... 826
The **java.lang.invoke** Subpackage ... 827
The **java.lang.ref** Subpackage ... 827

Basics of Annotation ... 827
Other Built-In Annotations ... 828
@Retention Annotation ... 828
@Documented Annotation ... 828
@Target Annotation ... 828
@Native Annotation ... 829
@Repeatable Annotation ... 829
@Retention Annotation ... 829
@Inherited Annotation ... 829
@Override Annotation ... 829
@Deprecated Annotation ... 830
@SuppressWarnings Annotation ... 830
@FunctionalInterface ... 830
@ SafeVarargs ... 830
Type Annotations and Pluggable Type Systems ... 831
Repeating Annotations ... 831
Retrieving Annotations ... 831
Immediate Solutions ... **832**
Creating a Package ... 832
Creating Packages that have Subpackages ... 832
Creating an Interface ... 833
Implementing an Interface ... 834
Extending an Interface ... 835
Using Interfaces for Callbacks ... 835
Performing Operations on a JAR File ... 836
Creating a JAR File ... 837
Getting the Contents of a JAR File ... 838
Extracting Files from a JAR File ... 838
Updating JAR Files ... 838
Reading from JAR Files in Code ... 838
Using **javac** to Get Classes from a JAR File ... 841
Letting Applets Get Classes from JAR Files ... 841
Marker Annotations ... 842
Single Member Annotations ... 843
Summary ... 844

Chapter 23: Working with Java Beans 845

***In Depth* 846**

What is Java Bean? 846
Advantages of Java Bean 847
Introspection 847
Design Patterns for Properties 847
Design Patterns for Events 848
Methods and Design Patterns 849
Using the BeanInfo Interface 849
Persistence 849
Customizers 849

***Immediate Solutions* 850**

Understanding Java Beans 850
Designing Programs Using Java Beans 850
Creating Applets that Use Java Beans 853
Creating a Java Bean 853
Creating a Bean Manifest File 855
Creating a Bean JAR File 855
Creating a New Bean 855
Adding Controls to Beans 855
Giving a Bean Properties 856
Design Patterns for Properties 862
Using Simple Properties 862
Adding a Color Property to SimpleBean 862
Using Indexed Properties 864
Using Boolean Properties 864
Designing Patterns for Events 864
Learning Methods and Design Patterns 865
Using the BeanInfo Interface 865
Using Feature Descriptors 865
Creating Bound Properties 865
Giving a Bean Methods 866
Giving a Bean an Icon 867
Creating a **BeanInfo** Class 867
Setting Bound and Constrained Properties 868

Implementing Persistence ... 870
Using the Java Beans API ... 870
Learning the Basics of an Event ... 872
Using the Java Beans Conventions ... 873
Using the Remote Notification and Distributed Notification ... 873
Using Beans with JSP ... 873
Summary ... 874

Chapter 24: Introducing JDBC ... 875

In Depth **... 876**

What does JDBC Do? ... 876
Components of JDBC ... 876
JDBC Specification ... 877
JDBC Architecture ... 877
The JDBC Package ... 878
The **java.sql** Package ... 879
The **javax.sql** Package ... 881
JDBC versus ODBC and Other APIs ... 883
Relation between JDBC and ODBC ... 883
Various Kinds of JDBC Drivers ... 884
Describing the Type-1 Driver ... 884
Describing the Type-2 Driver (Java to Native API) ... 885
Describing the Type-3 Driver (Java to Network Protocol/All Java Driver) ... 886
Describing the Type-4 Driver (Java to Database Protocol) ... 888
The JavaSoft Framework ... 888
Two-Tier and Three-Tier Models ... 889
The Structured Query Language (SQL) ... 890

Immediate Solutions **... 892**

Learning about the Driver Interface ... 892
Learning about the DriverManager Class ... 892
The Connection Interface ... 893
The Statement Interface ... 896
The PreparedStatement Interface ... 899
Comparing the Execution Control of the Statement and PreparedStatement ... 899
Describing the setXXX() Methods of the PreparedStatement Interface ... 901
Advantages and Disadvantages of Using a PreparedStatement Object ... 902

Using the PreparedStatement Interface ... 902
Create a PreparedStatement Object ... 903
Provide the Values of the PreparedStatement Parameters ... 903
Execute the SQL Statements ... 903
The ResultSet Interface ... 904
Describing the Methods of ResultSet ... 904
Using ResultSet ... 907
Moving the Cursor Position ... 908
Reading the Column Values ... 908
The ResultSetMetaData Interface ... 908
Describing the ResultSetMetaData Interface ... 909
Using the ResultSetMetaData Interface ... 910
Implementing JDBC Processes with **java.sql** Package ... 911
Understanding the Basic JDBC Steps ... 912
Processing the ResultSet ... 914
Using the ResultSet Object to Update Data ... 919
Using the ResultSet Object to Delete a Row ... 920
Using a ResultSet Object to Insert a Row ... 922
Closing the Statement ... 923
Closing the Connection ... 923
Using a Loop to Set Values in PreparedStatement Object ... 924
Returning Values for the executeUpdate() Method ... 924
Using the Interactive SQL Tool ... 926
Summary ... 926

Chapter 25: Images and Animation ... 927

In Depth **... 928**

Images ... 928
Different Formats of Images ... 928
The **ImageIcon** Class ... 928
The **java.awt.geom** Package ... 928

Immediate Solutions **... 930**

Using the **Image** Class ... 930
Creating an Image ... 930
Loading an Image ... 930
Displaying an Image ... 930
Drawing an **Image** Object ... 931

Loading an Image Object Using the **ImageObserver** Interface ... 935
Using the **MediaTracker** Class ... 937
Using the **ImageIcon** Class in an Applet ... 938
Getting Images ... 939
Drawing Images ... 940
Using **ImageObserver** Interface ... 942
Creating a Game Using the **java.awt.geom** Package ... 943
Collecting the Images ... 943
Organizing and Loading the Images in the Applet ... 944
Animating the Images ... 945
Finishing Up ... 947
Summary ... 950

Chapter 26: Java DB ... 951

In Depth **... 952**

Evolution of Java DB ... 952
Comparisons between Java DB and MySQL ... 953
Architecture of Java DB ... 954
Embedded Architecture ... 954
Client/Server Architecture ... 954
Compilation of SQL into Java Bytecode ... 955
Drivers of Java DB ... 955
Java DB URLs ... 956
Java DB Product Documentation ... 956
Derby Tools and Utilities ... 957
Installing and Configuring Java DB on Windows ... 957

Immediate Solutions **... 958**

Configure Environment Variables ... 958
DERBY_HOME ... 958
PATH ... 958
JAVA_HOME ... 958
Java DB Directory Structure ... 959
Creating Database in Java DB Using ij-Tool ... 959
Creating a New Database in Embedded Mode ... 959
Creating a New Database in Network Mode ... 961
Connecting with a Database ... 963
Connecting with a Database in Embedded Mode ... 963

Connecting with a Database in Network Mode 963
Disconnecting from a Database 964
Adding Tables in an Existing Database 964
Deleting Tables from an Existing Database 964
Adding Data into a Table 965
Displaying Content of a Table 965
Java DB Database Encryption 966
Using Java DB with Java Application 966
Summary 968

Chapter 27: Java FX 969

***In Depth* 970**

Working with JavaFX 970
JavaFX Applications 970
JavaFX Availability 970
Key Features of JavaFX 971
Application Development Scope of JavaFX 972
JavaFX Ensemble8 972
Modena 973
3DViewer 973
JavaFX Architecture 974
Scene Graph in JavaFX 974
Java Public APIs for JavaFX Features 975
Graphics System 975
Glass Windowing Toolkit 976
Threads 976
Pulse 976
Media and Images 977
Web Component 977
CSS 977
UI Controls 978
Layout 978
2-D and 3-D Transformations 979
Visual Effects 979
Introducing Scene Builder in JavaFX 979
Intended Audience 980
Key Features of JavaFX Scene Builder 980

Immediate Solutions 981
Creating Simple JavaFX Application 981
Creating JavaFX Application on NetBeans IDE 981
Using the UI Controls 989
Using Visual Effects 989
Using Canvas API 990
Summary 992

Glossary **993**

Index **997**

Bonus chapters available with the book for Download

Chapter 1: Java and XML—Using the Document Object Model **1**

Chapter 2: Java and XML—Using the Simple API for XML **31**

Chapter 3: Understanding RMI **57**

Chapter 4: Working with Servlets 3.1 ..... **83**

Chapter 5: Dynamic Java Scripting **137**

Chapter 6: Introduction to Groovy **159**

Chapter 7: MXBeans and JMX **189**

Chapter 8: Inclusion of Sound **213**

Chapter 9: Filing and Printing Documents **251**



Introduction

Thanks for buying the *Java 8 Programming* Black Book. The book covers Java 8 – a 2014-released Java version – in a comprehensive and accessible manner.

Java is a programming language that all novice programmers are willing to learn for building applications. Here this book helps by offering simple to complex Java programming concepts along with various programming examples. You will find programs on all Java concepts, which are meant for easy assimilation of a concept.

With the evolution of Java 8, various new concepts emerged, which are also part of this book. Its each chapter has been divided into *In Depth* and *Immediate Solutions* sections: where the *In Depth* section is about the theoretical understanding of one concept, the *Immediate Solutions* section is about the practical implementation of the same.

The most important feature of Java programming language is that it offers security; therefore, it is preferred over other programming languages. As most system applications are written in this language, it is only better that programmers be well versed with Java programming language.

About this Book

Java 8 Programming Black Book encompasses syntax to declare variables, as well as advanced object-oriented concepts. The book also deals with the programming implementation of the concepts explained. Using Java with XML, setting security permissions for applets, using the Java browser plugin, creating client/server connections over the Internet, creating Java Beans, connecting to databases, and multithreading have also been covered.

The book is divided into hundreds of topics – topics that are well spaced, easily accessible, each addressing a different programming issue, and each with an example showing just how it works. Here are just a few of those topics:

- Full Java 8 version 1.8 syntax
- Inheritance and inner classes
- Object-Oriented Programming (OOP)
- Abstract Windowing Toolkit (AWT)
- Generics
- Metadata
- Enumeration
- Autoboxing and Unboxing
- Buttons, checkboxes, and radio buttons
- Choosers, lists, and combo boxes
- Graphics, images, text, and fonts
- Menus, dialog boxes, and windows
- Progress bars, sliders, separators, and scrolling
- Image processing and tracking

- Java Swing
- Swing's pluggable look and feel
- All Swing components
- Swing text components
- Java collections
- Multithreading
- I/O Streams
- File handling
- Networking and sockets
- Split panes, editor panes, text panes, and more
- Trees and tables
- Java Beans
- MXBeans
- Packages, interfaces, and JAR files
- Java Server Pages (JSP)
- Reading applets from JAR files
- Security issues
- Java Database Connectivity (JDBC)
- Servlets
- Java DB
- Java FX

You will also come to know of some good online resources. In addition, there are some bonus chapters in the book on the topics like Java and XML, Groovy, Sounds in Java, and many more.

How to Make Use of This Book

In the book, since we are using Java 8, version 1.8, the given codes must be run in the same version; or else, you might get some strange errors. You can get this version of Java for you from the http://www.oracle.com/technetwork/java/javase/downloads/index.html URL.

You'll also need some way of creating Java programs. Such programs are just plain text files filled with Java statements and declarations. To create a Java program, you should have an editor program that can save files in a plain text format. See the topic "Writing Code: Creating Code Files" in Chapter 1 for details.

The JDK has all you need to create standard Java applets and applications, and also has an appletviewer for displaying applets at work.

Besides, we will also be using Beans Development Kit (BDK) and Java Servlet Development Kit (JSDK). There is also Java Server Pages (JSP) in the book, for which you will need a web-server-supporting JSP. If you want to work with the database programming of this book, you will need to create an ODBC data source on your machine. All the codes, images, and files used in the book can also be found online – not to mention a great number of other tools.

Conventions

There are a few conventions in the book that you should know in the first place. Some of them are as follows:

- When some code is new and needs to be pointed out, it'll appear shaded like this:

```
setLayout(null);
text1 = new TextField(20);
text1.setSize(200, 50);
text1.setLocation(20, 20);
add(text1);
```

- The book also gives many tips throughout for you to have more insight into the subject. Ex:

TIP

An untrusted applet is not allowed to read and write files at all. So unless you make specific security arrangements, you should not try to read or write files from applets. Therefore, for the sake of convenience, all the programs in this chapter are applications.

- You will also see notes for some additional information on the subject. Ex:

NOTE

It should be clear that there are situations where the Refresh response header can come in handy. Even though it is not a standard HTTP 1.1 header, it is something that can be considered a de facto standard.

- Every figure has a caption to maintain clarity. Ex:

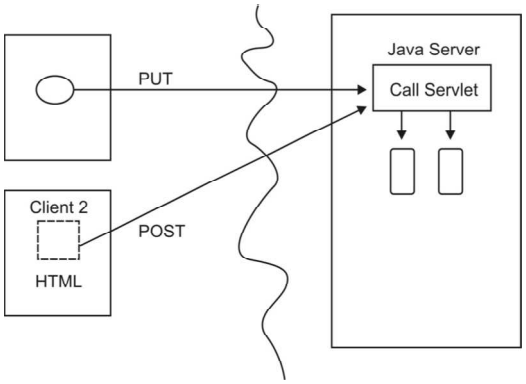


Figure 32.1: Showing Client/Server communication.

- The tables follow right below their reference. Ex:

Table 7.1: Constructors of the TextField class

Constructor	Does this
`TextField()`	It constructs a new text field.
`TextField(int columns)`	It constructs a new, empty text field with the indicated number of columns.
`TextField(String text)`	It constructs a new text field with the indicated text.
`TextField(String text, int columns)`	It constructs a new text field initialized with the indicated text and with the indicated number of columns.

Other Resources

Java comes with an immense amount of documentation – equal to and worth hundreds of books! This documentation is stored in linked HTML pages, and you must have a web browser to view and work with it. There are also many, many webpages out there on Java (a random Web search on Java turns up about 239,000,000 pages; in fact, searching for "Java tutorial" alone will turn up about 35,700,000 pages). Here are some other useful resources:

- The Java home page at http://www.oracle.com/technetwork/java/index.html
- The Sun Java tutorial at http://docs.oracle.com/javase/tutorial/
- The Java 8, version 1.8 documentation at http://docs.oracle.com/javase/8/docs/

The Black Book Philosophy

Written by experienced professionals, Black Books provide immediate solutions to global programming and administrative challenges, helping you to complete specific tasks, especially critical ones, which are not well documented in other books. Black Books help you understand a concept and then implement it practically by developing codes and applications based on that concept. The unique format of Black Books is structured to help you build your knowledge to solve problems and quickly master complex technical issues to become an expert. The discussion of complex topics has been split over many subheadings, each dealing with a particular aspect of that topic, enabling an easy and quick understanding.

Written and edited by the Content and Editorial teams at Dreamtech Press, this book is conceptualized to give you everything you need to know about Java 8.



1

Essential Java

If you need an immediate solution to:	*See page:*
Getting and Installing Java	15
Subfolders Created by JDK	16
Creating Source Files	16
Knowing Java's Reserved Words	17
Creating an Application	19
Compiling Code	20
Running Code	23
Running Code: Using Command-Line Options	25
Commenting Your Code	26
Import Java Packages and Classes	27
Finding Java Classes with CLASSPATH	28

In Depth

Welcome to our big book of Java programming. Herein, we'll cover as much Java programming as can be crammed into one book—in depth and in detail. We won't turn away from the more difficult issues because the aim of this book is to lay out all of Java for you, making it ready for use. If you are like us and have a few programming packages, you'll enjoy working with Java more than with others. And, we would like to presume that Java would be your choice for programming platform as you finish the book.

This first chapter covers the fundamental Java skills that you'll rely on in the coming chapters. In the next few chapters, you are going to see a large amount of Java syntax at work, but none will be of any use unless you can get Java running and create programs with it. That fundamental set of skills—creating and running Java programs—is the topic of this chapter, and you can put this information to work in the following chapters for testing the Java syntax we'll develop.

In this chapter, we are going to work through the mechanics of creating Java programs—from installation issues to writing Java code and from making sure your Java program can find what it needs to displaying simple output. These skills are the ones you can use in the coming chapters. The material in those chapters is all about the internals of writing a Java code, and this chapter is all about the rest of the process that makes the code run.

You might already be knowing much of the material in this chapter, in which case it will provide a good review (some of the material is bound to be new—after all, very few people know what all the Java compiler command-line switches do). If you've already got a working installation of Java and can write and run basic Java programs, you are already familiar with most of what you'll see in this chapter. Therefore, you can just skim through the following pages and continue with Chapter 2, where we start digging into Java syntax—the internals that really make Java work. Otherwise, you can work through the material in this chapter because it provides the foundation for the next several chapters to come. Here, we'll also make you familiar with the Application servers, Deployers, and various methods and constructors.

All about Java

Where did Java come from, and why is it so popular? Like other programming languages, Java filled a specific need of its time. For example, before Java appeared, C was an extremely popular language among programmers, and it seemed that C was the perfect one combining the best elements of low-level assembly language and high level languages into a programming language that fits into computer architecture well and that programmers liked.

However, C too had its limitations like the earlier programming languages had before it. As programs grew longer, C programs became more unwieldy, because there was no easy way to cut up a long C program into self-contained compartments. This meant that the code in the first line of a long program could interfere with the code in the last line, and the programmer had to keep the whole code in mind while programming.

To cut long programs into semi-autonomous units, Object-Oriented Programming became popular. With this, the motto was "divide and conquer". In other words, you could divide a program into easily conceptualized parts. For example, if you have a complex system that you use to keep food cold, you might watch the temperature of the food using a thermometer; and when the temperature gets high enough, you throw a switch to start the compressor for making the valves work so that the coolant circulates; then, you start a fan to blow air over the cooling vanes, and so on. That's one way to do it. However, another is to connect all those operations for making them automatic, wrapping the whole into an easily conceptualized unit—a refrigerator. Now, all the internals are hidden from your view, and all you have to do is put food in it or take it out of the refrigerator.

That's the way objects work; They hide the programming details from the rest of the program, reducing all the interdependencies that spring up in a long C program by setting up a well-defined and controllable interface that handles the connection between the object and the rest of the code. Now, you can think of the object in an easy way—for example, you might have an object that handles all the interaction with the screen, an object you call *Screen*. You can use that object in ways you'll see throughout this book to manipulate what it is intended to work on (in this case, the screen display). After creating the object, you know that the screen is handled by that

object and can put it out of your mind—no longer does every part of the code has to set up its own screen handling; you can just use the 'Screen' object instead.

When Object-Oriented Programming was added to C, it became C++, and the programmers had a new darling. C++ let programmers deal with longer programs and object-oriented code, which helped to solve many other problems as well. For example, supporting objects made it easier for the manufacturers that supply software to provide you with lots of prewritten code, ready to use. To create an object, you use a *class*, which acts like a template or cookie cutter for that object, that is, a class is to an object what a cookie cutter is to a cookie. In other words, you can think of a class as an object's type, much like a variable's type might be the integer type.

Because C++ supported classes, the software manufacturers could provide you with huge readymade libraries of classes, ready for you to start creating objects from. For example, one of the most popular libraries of C++ classes is the Microsoft Foundation Class (MFC) library that comes with Microsoft's Visual C++. Earlier, you wrote a Windows program in C, you needed about five pages of solid code just to display a blank window. However, using a class in the MFC library, you could simply create an object of the kind of window you wanted to use—with a border, without a border, as a dialog box, and so on. The object already had built-in functionality of the kind of window you wanted to create, so all it took to create that window was one line of code—just the line where you create the new window object from the class you selected.

Even more impressive was the fact that you could use an MFC class as a *base class* for your own classes, adding the functionality you want to that class through a process called *inheritance* in Object-Oriented Programming. For example, suppose you want your window to display a menu bar. You can *derive* your own class from a plain MFC window, adding a menu bar to that class to create a new class. In this way, you can build your own class just by adding a few lines of code to what the Microsoft programmers have already done. (Note that you'll see how Object-Oriented Programming works in depth in this book.)

All this seemed great to programmers, and C++'s star rose high. It appeared to many that the perfect programming language had arrived. What could be better? However, the programming environment itself was about to undergo a great change with the popularization of what amounted to an immense new programming environment—the Internet. And, that's what made Java so popular.

Java Appears

Java was not originally created for the Internet. The first version of Java appeared in 1991 and was written in 18 months at Sun Microsystems. In fact, it wasn't even called *Java* in those days; it was called *Oak* and was used internally at Sun.

The original idea of Oak was to create a platform-independent, object-oriented language. Many programmers were confining themselves to programming for the IBM PC at that time, but the corporate environment can include all kinds of programming platforms—from PC to huge mainframes. The driving inspiration behind Oak was to create something that could be used on all those computers (and now that Java has been popularized by the Internet, a huge and increasing number of corporations are adopting it internally instead of C++ for just that reason—in fact, some versions of C++, such as Microsoft's C#, are now being modeled after Java). The other motivation behind the introduction of Oak was to bring a language in the picture that could be used in consumer electronics.

Oak was renamed Java in 1995, when it was released for public consumption, and it was almost an immediate hit. By that time, Java had adopted a model that made it perfect for the Internet—the bytecode model. At present, Sun has been taken over by Oracle; therefore, the platforms of Java are the products of Oracle now.

All about Bytecodes

A Microsoft Visual C++ program is large, typically starting off at a minimum of 5 MB for a full MFC program and that doesn't even count the Dynamic Link Libraries (DLLs) that the Windows platform needs to run Visual C++ programs. In other words, C++ programs are fully executable on your computer as they stand, which means they have to be large in size. Imagine how cumbersome it would be to download all that as part of a Web page for letting that page do something interactive on your computer.

Java programs, on the other hand, are constructed differently. Java itself is implemented as the Java Virtual Machine (JVM), which is the application that actually runs your Java program. When JVM is installed on a

computer, it can run Java programs. Java programs, therefore, don't need to be self-sufficient, and they don't have to include all the machine-level code that actually runs on the computer. Instead, they are *compiled* into compact bytecodes, and it's these bytecodes that JVM reads and interprets to run your program. When you download a Java applet from the Internet, you are actually downloading a bytecode file.

In this way, your Java program can be very small, because all the machine-level code to run your program is already on the target computer and doesn't have to be downloaded. To host Java on a great variety of computers, Sun only had to rewrite JVM for wording on those computers. Because your program is stored in a bytecode file, it will run on any computer on which the JVM is installed.

As Java programs were originally supposed to be interpreted by the JVM, that is, they were executed bytecode by bytecode, interpretation was a slow process. For that reason, Java introduced the Just-In-Time (JIT) compiler, which is built into JVM. The JIT compiler reads sections of the bytecode and compiles them dynamically into a machine-level code. Java performs runtime checks on different code sections; therefore, the whole program is not compiled at once, resulting in the a faster execution of the programs.

Using bytecodes means that the Java programs are very compact, making them ideal for downloading over the Internet. Moreover, running such programs with the JVM rather than downloading full programs provides security.

Features of Java

Java is a simple, robust, platform-independent, and portable programming language. It is designed to meet the real-world requirements with its following features:

- Simple
- Secure
- Portable
- Robust
- Multithreading
- Platform independent
- Distributed
- Dynamic

Simple

Java is simple to learn and use. In Java, it is easy to write and debug programs because Java does not use pointers. Moreover, it provides bug free system because to strong memory management and also has automatic memory allocation and de-allocation system. Unlike other programming languages, which provide dozens of complicated ways to perform a simple task, Java provides a simple and clear approach to achieve a given task. If a programmer understands the basic concepts of OOP, it becomes easier for him/her to learn Java.

Secure

In today's world, everybody needs safety and security. Because of to the threat of hackers, people feel unsafe while performing commerce over the Internet. To overcome such fears, Java provides safety and security features.

While using Java-compatible browser, everyone can safely download Java applets by relaxing their minds from the fear of virtual infection or malicious intent. The Java programs are confined to the Java runtime environment, they can access only that part of your computer that is required for program execution. The data outside Java runtime environment cannot be accessed by them and therefore, downloading a Java application through the Internet will not harm your computer. That's why Java is a secure language.

When Java executes a program, the JVM can strictly monitor what goes on, which makes it great for the Internet applications. You are already aware that security has become an extremely important issue on the Internet, and Java rises to the task of taking care of the same. JVM can watch all that a program does; and if it does something questionable, such as trying to write a file, it can prevent that operation. That alone makes Java more attractive than C++, which has no such restrictions, for the Internet.

You can also tailor Java security the way you like it, which offers a very flexible solution. As you'll see in this book, you can now specify, on a program-by-program basis, just what privileges you want to give to the downloaded code. You can now also "sign" your Java code in a way that shows it comes from you without any malicious modifications. We'll take a look at all this and more in this book.

In a nutshell, Java has a winning combination for the Internet—Java programs are small, secure, platform-independent, object-oriented, and powerful. They also implement other features that programmers like. Java programs are often simple to write compared to C++. The end result is that you can write your Java program once, and it can be easily downloaded and run on all kinds of machines—the perfect recipe for the Internet! That's the reason Java has soared so high.

Portable

In the distributed world of the Internet, an application developed with the help of a programming language might be accessed on various computers having different kinds of operating system. But, it's not guaranteed that the application is portable, that is, it runs successfully on all operating systems. To overcome this portability issue, Java introduces a concept of what is known as bytecode.

Bytecode is a set of instructions generated by the Java compiler while compiling a Java program. In other programming languages, a program is compiled into an executable code; but in Java, a program gets compiled into an intermediate code called bytecode. This bytecode then gets executed by Java runtime system called the JVM. So, in order to run your Java program, you should have only JVM installed. It is also considered to be the interpreter of bytecode. In this way, Java has solved the problem of portability.

Robust

Generally, programs fail because of the following two reasons:

- **Memory Management**—Lack of memory management for allocating and de-allocating memory for objects
- **Exceptional Conditions**—Occurrence of unexpected exceptions

Considering these reasons, Java was designed to provide the ability to create robust programs. Therefore, Java compels the programmers to resolve mistakes during compilation. Moreover, it also frees the programmer from the worries of common causes of programming errors as the program code is verified during compilation.

Java also eradicates the problem of memory management by performing automatic garbage collection. It manages the memory allocation and de-allocation for unused objects.

Multithreading

Java is a programming language that is designed for the distributed environment of the Internet; and for that, the concept of multithreading is implemented. This feature helps you write interactive programs, wherein multiple tasks can be performed simultaneously, thus making Java a robust programming language.

Java is inherently multi threaded as there can be multiple executing threads in a single Java program. For example, a Java programmer can run three Java applets on the same page, provided that each applet gets equal time from the CPU.

The multi threaded feature of Java makes it very responsive and robust. It is very important to note that as Java is highly multi threaded, it is difficult to port it to architectures, such as Windows 3.1 or PowerMac, which do not natively support multithreading.

Platform-Independent

It is one of the most important features of Java as it is the first programming language that is not bound to any specific operating system. In other words, Java programs can be executed anywhere on any system. Before the evolution of Java, no other programming language was platform-independent, and it can therefore be said that Java is a revolutionary technology. Apart from cross-platform in source form, Java is also platform-independent in complied binary form.

A Java program does not execute natively on the host machine; instead, a Java interpreter reads the bytecode and executes the corresponding native machine instructions. The Java interpreter is a special naive program that reads the bytecode, an intermediate form after compilation. Therefore, in order to port a Java program to a new

platform, you need the interpreter and some of the library routines. Moreover, the Java compiler is also written in Java, and the bytecodes are precisely defined, which remain the same on all platforms.

Distributed

Java is a distributed language as it can be used to create applications for communicating over the network. Java can communicate over the network because it supports `TCP/IP` (Transmission Control Protocol/Internet Protocol). The `TCP/IP` protocol is a set of network communication protocols.

Dynamic

During the runtime of a Java program, the relevant information that is used to verify and resolve access to objects is required. This concept of providing runtime information is referred to as dynamically linking the code. This feature of Java adds strength to the applet environment, in which small fragments of bytecode are dynamically updated on a running system.

Java Environment

The Java environment comprises numerous development tools, classes, and methods. The development tools for Java are provided as a part of the system known as Java Development Kit (JDK). Moreover, the Application Programming Interface (API) comprises the classes and interfaces used in Java programming. In this section, we discuss the following:

- Java Development Kit
- Java Platforms
- Java Virtual Machine
- Java API
- Java Programs

Java Development Kit

Keeping in mind the developers, Sun introduced JDK originally, which is now an Oracle product. JDK provides a collection of the Java tools, which are used while developing and running Java programs. The kit comprises the following development tools:

- **java**—Serves as a Java interpreter used to run Java applets and applications by reading and interpreting bytecode files
- **javac**—Serves as a Java compiler used to translate Java source code into bytecode files
- **javacdoc**—Creates HyperText Markup Language (HTML) documentation for Java source code files
- **javap**—Serves as a Java disassemble used to convert bytecode files into a Java program description
- **jdb**—Serves as a Java debugger used to find errors in Java programs
- **appletviewer**—Facilitates to run Java applets
- **jar**—Serves as an archive used to package related class libraries into a single executable JAR file and also helps to manage the JAR files
- **javah**—Serves as the C header and stub generator used to write native methods

Java Platforms

Oracle provides various Java editions to develop and run Java programs. The Java platform is not bound to any one processor or operating system, but the execution engine (virtual machine) and a compiler can run identically on any operating system. The following subsections are the different editions of the Java platform.

What is Java Standard Edition (Java SE)?

The Java SE Edition of Java is the most commonly used among the three Java editions. It is used for developing desktop and console-based applications. It consists of:

- A runtime environment

- Sets of APIs to build a wide variety of applications, comprising standalone applications that can run on various operating systems, such as Microsoft Windows, Linux, and Solaris

What is Java Enterprise Edition (Java EE)?

Java EE is a platform primarily used for building server-side applications by using its convenient component-based approach. It is a platform-independent, Java-centric environment from Oracle, which is meant for the development and online deployment of Web-based Enterprise applications. The Java EE platform consists of a set of service—APIs and protocols. These services provide the functionality for developing multi tiered, Web-based applications.

Java EE is a Java-based, runtime platform created by Sun. It is now owned by Oracle after the acquisition of Sun by Oracle on January 27, 2010. Java EE is used for developing, deploying, and managing multi tier server-centric applications on an enterprise-wide scale. It builds on the features of Java SE and adds distributed communication, threading control, scalable architecture, and transaction management. A strong competitor to the Microsoft's .NET Framework, Java EE is a Java platform designed for the mainframe-scale computing, typical of large enterprises. For the simplification of application development in a thin client-tiered environment, Sun designed Java EE in collaboration with partners such as IBM.

Some of the key features and services of Java EE are as follows:

- At the client tier, it supports pure HTML as well as Java applets or applications. It relies on Java Server Pages and Servlet code to create HTML or other formatted data for the client.
- Enterprise JavaBeans (EJB) provides a layer for storing the platform's logic. The EJB server provides functions such as threading, concurrency, security, and memory management. These services are transparent to the author.
- Java Database Connectivity (JDBC), which is the Java equivalent to Open Database Connectivity (ODBC), is the standard interface for Java databases.

The Java Servlet API increases consistency for developers without the need of a Graphical User Interface (GUI).

What is Java Micro Edition (Java ME)?

It enables you to build Java applications for micro-devices, which include handheld devices, such as cell phones and Personal Digital Assistants (PDAs), and other similar devices with limited display and memory support. It is a competitor to Microsoft's .NET Compact Framework.

The devices that you use nowadays, such as cell phones, PDAs, televisions, and set-top boxes, are often powered by Java ME. This is because Java ME provides a robust and flexible environment to enable the applications meant to run on devices that are low on resources, such as processor and memory. The Java Community Process, which comprises the experts from the leading device manufacturers, defined the standard set of Java APIs that are part of the Java ME environment along with the JVM. The same sets of APIs along with the JVM are also included in the other Java platforms that include Java SE, Java EE, and Java Card environments.

The strength and advantages of the Java technology can be provided to the end user of the embedded consumer devices by using Java ME in such devices. The inclusion of a flexible user interface, a robust security model, a variety of built-in network protocols, and an extensive support for networked and offline applications that are automatically downloadable makes Java ME a versatile platform for a wide range of consumer devices. The applications written for the Java ME specifications are also inherently platform-independent and can therefore be used on a variety of devices even though written for one.

The Java ME architecture includes various configurations, profiles, and optional packages for the developers to choose from. The chosen packages and configurations can be combined to build a complete Java runtime environment to go in rhyme with the target device and the target market. The selection is optimized for the processing power, memory, and I/O capabilities of any related category of devices. This eventually results in a common platform that benefits from each category of devices by providing an engaging user experience.

The Java ME configuration includes a JVM and a set of class libraries. The number of libraries is kept to a bare minimum because it is meant to be executed on a device with a modest hardware configuration. They then provide the basic functionality for a particular category of devices having common characteristics. There are two

Java ME configurations: the Connected Limited Device Configuration (CLDC) and the Connected Device Configuration (CDC).

Java Virtual Machine

In all programming languages, a compiler translates source code into machine code for a specific computer. In Java, a compiler translates the Java source code into an intermediate code known as bytecode for a virtual machine, known as JVM. Bytecode is the machine language for JVM and is not machine-specific. Instead, the machine-specific code is generated by the Java interpreter, which serves as a mediator between virtual machine and real machine. Figure 1.1 shows the process of compilation and conversion of bytecode into machine code:

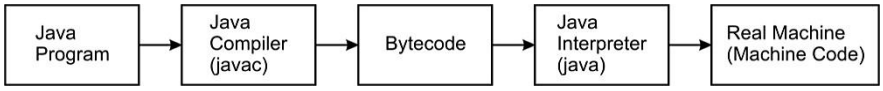


Figure 1.1: Displaying the Process of Compilation and Execution in Java

In order to run a Java program on a different computer, you need a Java interpreter for Java bytecode. Although Java bytecode interpreter is required depending upon the type of operating system in use, once a computer has an interpreter, it can run a Java bytecode. In other words, the most important feature of Java is that the same compiled program can run on any operating system, as shown in Figure 1.2:

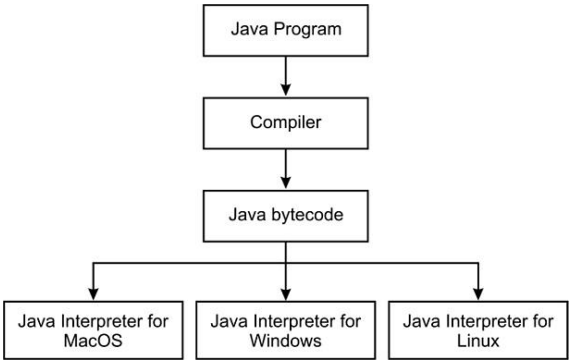


Figure 1.2: Displaying the Process of Running a Program on Any Operating System

Java API

It is a collection of classes, interfaces, and methods provided in the form of Java packages. In other words, it is a large collection of already created classes and interfaces, which provide many useful capabilities, such as Graphical User Interface (GUI), date, time, and calendar capabilities. The categorization of Java API in the form of Java packages is as follows:

- Application Programming Packages
  - **java.lang**—Provides classes that are fundamental to the design of the Java programming language
  - **java.util**—Provides legacy collection classes, event model, collection framework, date and time capabilities, internationalization, and other utility classes, such as string tokenizer
  - **java.io**—Provides classes for system input and output through data streams, serialization, and the file system
- Applet and Network Programming Packages
  - **java.awt**—Provides classes for creating user interface and painting graphics and images
  - **java.applet**—Provides classes that are necessary for creating an applet and those that are used to communicate with its applet context
  - **java.net**—Provides classes that are used for implementing networking in Java programs

Java Programs

A Java programmer can create either a Java application or a Java applet depending upon the requirement. In Java, you create the following types of program:

- Applications
- Applets

We'll use the term program in this book to refer to both applets and applications. Now, let's discuss each of them in detail.

Applications

A Java application is a program that is created by using Java programming language. Java applications are console-based, Character User Interface (CUI)-based, and GUI-based standalone applications. Once you become used to creating applications, you will find that creating an application by using Java is also as simple as creating an application with any other programming language. In order to execute a Java application, you need to perform the following steps:

- Compile the Java source code by using `javac` compiler to translate the source code into bytecode
- Execute the bytecode by using the Java interpreter

Apart from applications, you can also create applets in Java.

Applets

An applet is a program written in the Java programming language that can be included in an HTML page in the same way as an image is included. An applet can be used in both `static` and `dynamic` web pages to either display content or share information through the pages. You need to import the `java.applet` package that provides classes to either create an applet or allow an applet to communicate with its applet context.

The applet context is an application that is responsible for loading and running applets. For example, the applet context could be a web browser or an applet development environment. In other words, an applet is a Java application designed to be transmitted over the Internet and is executed by a Java-compatible web browser. After understanding the types of Java program, let's create a simple Java program. But to do this, you need to create a source code file by using a text editor first. After creating the Java program, you need to compile it by using the Java compiler `javac` and execute by using the Java interpreter `java`. The compiler generates a .class file by translating the source code into bytecode, which is then executed by the Java interpreter.

What are Application Servers (Tomcat, JRun, WebSphere, WebLogic, etc.)

An Application server is a server program running on a computer in a distributed network that provides the business logic for an application program. Frequently viewed as part of a three-tier application, this server consists of a GUI server, an application (business logic) server, and a database and transaction server. The server encapsulates the Java program code that executes within an Evolution account. Application servers are provided by third-party software vendors and are integrated into the `myEvolution` hosting control center.

In addition to the application logic, it contains the services that access the resource manager, such as a database. Clients or other services that may or may not reside on the same physical machine call the services. There are a number of Application servers used by various Java applications. Some examples are as follows:

- **Tomcat**—The Tomcat server is a Java-based Web application container that was created to run Servlets and Java Server Pages in Web applications. Tomcat is the Servlet container that is used in the official Reference Implementation for the Java Servlets and Java Server Pages technologies. The Java Servlets and Java Server Pages specifications were developed by Sun under the Java Community Process. It is developed in an open and participatory environment and released under the Apache Software License. The Tomcat is intended to be a group of the best-of-the-best developers from around the world.

- **JRun—**JRun is an Application server of Adobe that is based on Oracle's Java Enterprise Edition. It consists of Java Server Pages, Java Servlets, Enterprise Java Beans, Java Transaction Service (JTS), and Java Messaging Service (JMS).

  JRun works with the most popular web servers, including Apache, Microsoft's Internet Information Server (IIS), and any other web server that supports Internet Server Application Program Interface (ISAPI) or the Web's Common Gateway Interface (CGI).

  JRun comes in four editions—Developer edition, Professional edition, Advanced edition, and Enterprise edition. The Developer edition consists of the full JRun package, but it is meant for development use only and is limited to three concurrent connections. The Advanced edition is designed to deploy JSP and Servlet applications in a clustered-server environment. Corporates that host Servlet and JSP-based Web applications from a single server use the Professional edition. Companies that build and deploy e-commerce Java applications use the Enterprise edition.

- **WebLogic—**The WebLogic Platform delivers application infrastructure technology in a single, unified, easy-to-use platform for application development, deployment, and management. The Oracle WebLogic Platform increases productivity and lowers total cost of ownership for IT organizations as it provides a unified, simplified, and extensible platform for application development, deployment, and management.

  WebLogic is the first platform to address the need for asynchronous Web services, which is a critical aspect of application-to-application communication.

  WebLogic gives developers the tools necessary to integrate different applications and platforms for providing full leverage to organization in terms of data and software investments. It also provides an enterprise class framework that ensures reliable, scalable, available, and secure applications.

  To integrate different applications and platforms, the developers require Deployers for deploying Web and EJB applications in the server environment.

- **Deployers—**The Deployers are responsible for deploying EJB applications and Web applications into the server environment. They are not responsible for deploying an Application client archive or a resource adapter archive but may be responsible for additional configuration of these components.

  Packaged as a part of Java EE EAR files, these archives are not considered when the Enterprise applications are deployed. These are a part of Java EE applications but do not follow the runtime activation process that Web application and EJB containers go through during deployment.

  The Application client archives operate within the content of a Java EE container and are not deployed into an Application server. The Application client program runs standalone, and the Deployers are not responsible for configuring the container environment for these programs.

  The resource adapter archives are the simple libraries that are dropped into a valid JCA implementation.

  The Deployers produce container-ready EJB applications, Web applications, applets, and Application clients that have been customized for target environment of the Application server. Some of the commonly used Deployers are Ant, Junit, Cactus, and Maven.

  - **Ant—**Apache Ant is a Deployer, which is a Java-based tool. Ant is like Make but without Make's wrinkles. Unlike Make, Ant uses Java classes instead of writing shell commands.
  - **JUnit—**Developed by Erich Gamma and Kent Beck, JUnit is an open source software—a testing framework. The developer who implements unit tests in Java uses it. It is a simple framework to write the repeatable tests. JUnit is an instance of xUnit architecture for unit testing framework.
  - **Cactus and Maven—**The Apache Cactus and Maven, used as Deployers, are Java-based applications that automatically start containers, run tests, and stop the containers.

    Cactus is a simple framework for testing server-side Java code. Its purpose is to lower the cost of writing tests for server side codes.

    Maven is based on the concept of Project Object Model (POM), which provides a well-defined project structure, well-defined development process, and cohesive body of documentation that keep the developer updated regarding the project progress.

Program Design

Program design in Java is not necessarily an easy task. Good programming design involves a number of overall aspects, and it's worth taking a look at some of them in this chapter before we start digging into Java syntax in depth.

In fact, one of the most important aspects of creating a new application is designing that application. Poor choices can end up hampering your application through many revisions of the product. Many books are available on program design.

Microsoft, which should know something about it, breaks the process into four areas:

- **Performance**—The responsiveness and overall optimization of speed and resource use
- **Maintainability**—The ability of the application to be easily maintained
- **Extensibility**—The ability of the application to be extended in well-defined ways
- **Availability**—How robust the implementation of the application is and how available it is for use

Let's take a quick look at these four areas now.

Performance

Performance is a design issue that's hard to argue with. If the users aren't getting what they want from your application, that's clearly a problem. In general, performance depends on the users' needs. For some people, speed is essential; for others, robustness or efficient use of resources is what they are looking for. Overall, the performance of an application is an indication of how well it responds to the users' needs. Here are some general aspects of performance that you should consider while writing Java programs:

- Algorithm efficiency
- CPU speed
- Efficient database design and normalization
- Limiting external accesses
- Network speed
- Security issues
- Speed issues
- Use of resources
- Web access speed

We'll get to more specifics of performance throughout the book.

Maintainability

Maintainability is the measure of how easily you can adapt your application to future needs. This issue comes down to using good programming practices, which we'll talk about throughout the book. Much of this is common sense—simply keeping future coding needs in mind as you write your code. Some major issues in the "best programming" arsenal include the following:

- Avoid deep nesting of loops and conditionals
- Avoid passing global variables to procedures
- Be modular when you write code
- Break code into packages
- Document program changes
- Give each procedure only one purpose
- Make sure that your application can scale well for large tasks and large number of users
- Plan for code reuse
- Program defensively

- Use access procedures for sensitive data
- Use comments
- Use consistent variable names
- Use constants instead of "magic" numbers

Extensibility

Extensibility is the ability of your application to be extended in a well-defined and relatively easy way. It is usually a concern only with larger applications, and it often involves an entire interface specially designed for extension modules. In fact, Java itself is designed to be extended by using the Java Extension Framework.

Availability

Availability is the measure of how much time your application can be used for in comparison to the time users want to use it for. This includes everything, from the application not freezing up when performing a long task (at the least, giving the user some feedback of the operation's status) to working with techniques and methods not likely to hang, to making backups of crucial data, and to planning for alternate resource use—if possible—when access to a desired resource is blocked.

Overall, the design process is one that involves quite a bit of time. In fact, the whole development cycle is the subject of quite a few studies—you may be surprised to learn that when field testing, in-house testing, planning, designing, and user interface testing are added up, some studies allocate as little as 15 percent of total project time to the actual coding.

So much has been written about the development cycle of software that we won't go into more details here. But, it's worth noting that programmers shouldn't short-change the crucial design steps because it can lead to more problems in the long run than time saved in the short run.

New Features of Java SE 8

Java SE 8 is a major release that brought many new features and enhancements in Java programming language. Let's have a brief introduction of some noteworthy features of Java SE 8.

- **Java programming language—** Following are new changes in Java programming language:
  - Lambda expressions are introduced as a new language feature. They allow you to pass functions as a method argument or treat code as data.

    Following is the syntax of lambda expression:

```
view sourceprint?1 (argtype arg...) -> { return some expression.. probably using these
arguments }
```

  - Method references help in reading lambda expressions for methods that already have a name.
  - Default methods enable new functionality to be added to the interfaces of libraries and ensure binary compatibility with code written in older versions of those interfaces.
  - Repeated use of annotations enables you to apply the same annotation type more than once to the same type of declaration.
  - A new method, `String.join()`, is added to join the strings. For example:

```
String abc= String.join("String1 ", "Java", "8");
```

    The preceding code will be evaluated as:

```
String1 Java 8
```

- **Java compiler—** There is a new feature of using the -parameters option with javac command – to store formal parameter names and enable the Reflection API to retrieve them. Let's create the following class:

```
import java.lang.reflect.Method;
import java.lang.reflect.Parameter;

public class ParameterNames {
     public static void main(String[] args) throws Exception {

          Method m = ParameterNames.class.getMethod( "main", String[].class );
```

```
            for( final Parameter parameter: m.getParameters() ) {

                System.out.println( "Parameter: " + parameter.getName() );

            }

        }

    }
```

If you compile the previous class without using –parameters argument and then run this program, you will see the output as:

```
Parameter: arg0
```

However, if you compile the previous class with the –parameters argument and then run this program, you will see the output with the actual name of parameter that you have passed.

```
Parameter: args
```

- **Java libraries—** Following changes have taken place in Java libraries features:
  - **Streams—** A new package, java.util.stream, provides a Stream API to support functional-style operations on streams of elements. Stream API can be integrated with Collections API to perform bulk operations on collections, such as sequential or parallel map-reduce transformations.
  - **Date/Time API (JSR 310) —** Java 8 provides a new Date-Time API (JSR 310), which provides a new package, called java.time. This package contains all the classes for date, time, date/time, time zones, instants, duration, and clocks manipulation. Some examples of classes contained in this package are Clock, LocaleDate, LocalTime, and Duration. The Clock class provides the access to the current instant, date and time using a time zone. It can be used in place of using the `System.currentTimeMillis()` and `TimeZone.getDefault()` methods. The LocaleDate class holds only the date part and the LocaleTime class holds only the time part without a time-zone in the ISO-8601 calendar system. The Duration class, finally, provides an amount of time in terms of seconds and nanoseconds.
  - **Nashorn JavaScript engine—** Java 8 provides a new Nashorn JavaScript engine that allows you to develop and execute certain kinds of JavaScript application on JVM. The engine works like javax.script.ScriptEngine and permits Java and JavaScript interoperability.
  - **Parallel Arrays—** Java 8 provides various new methods allowing parallel arrays processing. For example, the `parallelSort()` method significantly speeds up the sorting on multicore machines.
  - **Concurrency—** Java 8 adds new methods to the `java.util.concurrent.ConcurrentHashMap` class that support aggregate operations on the basis of the newly added streams facility and lambda expressions. A new class, `java.util.concurrent.locks.StampedLock`, has been added to provide a capability-based lock with three modes for controlling read/write access. In addition, some classes are added to the `java.util.concurrent.atomic` package, which includes DoubleAccumulator, DoubleAdder, LongAccumulator, and LongAdder.
- **Java tools—** Java 8 provides a new set of command line tools, which include Nashorn engine (jjs command) and class dependency analyzer (jdeps command). The role of these tools is described as follows:
  - **Nashorn engine—** The jjs command accepts a list of JavaScript source code files as arguments and runs them. For example, if a file, myfunc.js, is created with the following content:

```
function demofunc() {

  return 1;

};
print( demofunc() + 1 );
```

Now, the jjs command is used to execute the myfunc.js file:

```
jjs myfunc.js
```

The output of the preceding code will be:

```
2
```

- **Class dependency analyzer—** The jdeps command is used to show the package-level or class-level dependencies of Java class files. It accepts .class file, a directory, or JAR file as an input, and displays the dependencies to the console.

- **Java runtime (JVM)—** The PermGen space has been replaced with Metaspace (JEP 122). The JVM options,XX:PermSize and -XX:MaxPermSize, are replaced with -XX:MetaSpaceSize and XX:MaxMetaspaceSize, respectively.
- **JDBC—** The JDBC-ODBC Bridge has been removed. In addition, JDBC 4.2 has been introduced with various new features. Some of these are as follows:
  - Addition of REF_CURSOR support
  - Addition of java.sql.DriverAction interface
  - Addition of security check on deregisterDriver method in the DriverManager class
  - Addition of the java.sql.SQLType interface
  - Addition of the java.sql.JDBCType enum

NOTE

You would learn about these features of Java SE 8 in detail with their implementation in other chapters of the book.

Immediate Solutions

Getting and Installing Java

The Big Boss (BB) gives you a call—as usual — at the last minute. You have 20 minutes to write a new Web page that gives users an overview of your company's products. What are you going to do? Knowing how well Java works in cases like this, you select Java as your language of choice to get the task done. Of course, you've got to make sure you have it before you can use it.

It's time to download and install Java, which means downloading the Java Software Development Kit (SDK). Currently, you can find it at `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`. The kit gets downloaded in the form of an executable application file. Next, you can follow the instructions on the Oracle's website in order to install JDK 1.8. We'd love to be able to provide the actual installation instructions here, but that's one of the biggest pitfalls a Java book can fall into, even the one that's designed to be as complete as possible. We have been writing about Java ever since it first came out, and it turns out that the actual installation instructions are often very volatile. As these instructions are changed quite frequently, the ones we have provided in the previous books instantly became obsolete. For that reason, the absolutely best thing you can do is to see how Sun wants you to install the SDK; therefore, you should refer to the installation instructions as posted on the Java site. The installation process has been getting easier with every version and beta of Java, and now, it typically involves only running the file you've downloaded.

One thing you should be sure to do, as indicated in the Sun installation instructions, is to ensure your machine can find the Java tools, including the Java compiler itself. To do that, verify that the Java bin directory is in your computer's path. For example, in Windows, usually the bin directory path is `C:\Program Files\Java\jdk1.8\bin` for SDK 1.8.

TIP

The Java Software Development Kit was called the Java Development Kit, JDK, in earlier Java versions. In fact, Oracle often seems to be of two minds here—sometimes, you'll see references to the SDK and sometimes, you'll see references to JDK. This is one of those times—although the kit is now called the SDK, it's installed by default into a directory named \jdk1.8.

In Windows 7, you can follow these steps to configure Java 8 on your system:

- Click the Windows icon on the taskbar, right click the Computer, and select the Properties option. You can also reach the same location by opening the Control Panel | System and Security | System.
- Click the Advanced system settings option in the System Properties window. This opens the System Properties dialog box.
- Click the Environment Variables button to open the Environment Variables dialog box.
- Edit the `PATH variable`, setting the way you want. You may append the new path to the already existing `PATH` variable or create a new `PATH` variable. Finally, click OK. Note that if you are appending the `PATH` variable, the different paths must be separated by semicolons (;).

You can also set the `PATH` variable temporarily with the help of the command line. The path shall remain set only till the command line is open; once you exit from the command line, the `PATH` variable set also gets erased. The following command is used to temporarily set the `PATH` variable:

```
SET PATH= C:\Program Files\Java\jdk1.8\bin;
```

When the bin directory is in the path, you'll be able to use the Java tools directly from the command line, instead of having to preface them with a pathname each time you want to use them on the command line.

What about CLASSPATH?

Java veterans will wonder about the environment variable named `CLASSPATH` when they install Java SDK. The `CLASSPATH` variable, as you'll soon see in this chapter, tells Java where to find compiled bytecode files—both the ones you create and the ones required by the system that come with the SDK itself. `CLASSPATH` has been the focus of a great deal of confusion when working with Java, and we are glad to say that Sun has made things easier.

When you install the SDK, you don't have to worry about setting the CLASSPATH now because the SDK will know where to find its own installed libraries. However, if you want to search other custom bytecode files when compiling a file, you'll have to set the CLASSPATH yourself. You'll see how to do this when we will discuss compiling programs in this chapter. (There are two ways to indicate to the Java compiler where to find bytecode files you particularly want to search—by setting the CLASSPATH environment variable and by using the -CLASSPATH compiler switch.)

The current version of the SDK does not require CLASSPATH to be defined; and if it has been defined by some other Java version or system, it is likely to create problems. If you want to keep the CLASSPATH environment variable for any reason, you need to use the command line option to define the CLASSPATH temporarily.

Subfolders Created by JDK

After installing JDK, several folders get created on your hard drive. The location of these folders formed depends on the particular system; however, in general, the JDK folder is formed under Program Files\Java on your boot drive. The name of the JDK root folder also includes the version you have installed. The different subfolders created under the JDK root folder and additional files are described in Tables 1.1 and 1.2, respectively:

Table 1.1: The different subfolders created in the JDK root folder

Folder	Description
bin	The compiler and other Java development tools
demo	Demo programs from which you can learn various Java features
include	Files required to integrate Java with other languages
jre	Runtime Environment files
lib	Library files, including the Java API class library
sample	Samples, with source code, of programming for certain Java API's
db	Oracle's distribution of the Apache Derby relational database

Table 1.2: Additional files in the JDK root folder

File	Description
README.html	The Java readme file in the HTML format
THIRDPARTYLICENSEREADME.txt	A license file for third parties, regarding usage and redistribution of JDK
LICENSE	The Java license that you agreed to when you downloaded JDK
COPYRIGHT	The copyright notice about all the copyright and export laws applied by Sun
register.html	For free register of the JDK installed with Oracle
release	It contains details such as Java version, OS name, and OS version
register_ja.html and register_zh_CN	These HTML files are, respectively, the Japanese and Chinese versions of register.html
Src	A .zip file that contains the code for Java API classes

Creating Source Files

The Design Team Coordinator (DTC) calls to congratulate you on getting Java installed. You accept the accolades gracefully. "So, what programs have you been writing?" the DTC asks. "Hmm!" you think. "Programs?"

Java programs are just plain text files made up of Java statements and declarations, and we'll start investigating them in the next solution. To create a Java program, you should have a text editor or word processor that can save files in plain text format.

Saving text in plain text format is a simple achievement that's beyond many fancy word processors. You might have trouble with word processors, such as Microsoft Word, although you can save plain text files with Word

using the File | Save As dialog box. The general rule is that if you can type the file out at the command line (note that that's DOS on DOS- and Windows-based computers) and don't see any odd, non-alphanumeric characters, it's a plain text file. The real test, of course, is whether the Java compiler, which translates your program into a bytecode file, can read and interpret your program.

In addition, your programs should be stored in files that have the extension "`.java.`" For example, if you are writing an application named `App`, you should store the actual Java program in a file named `App.java`. You pass this file to the Java compiler to create the actual bytecode file, as you'll see in the next few pages.

So far, so good—we have done the selection of the editor or word processor down. Now, how about writing some code?

Knowing Java's Reserved Words

A Novice Programmer (NP) appears and says, "Java is acting all funny—I want to name a variable "`public`," but it's giving me all kinds of problem." "That's because `public` is one of the keywords that Java reserves for itself as part of the Java language," you say. "Rats," says the NP.

When you are writing a Java code, you should know that Java reserves certain words for itself as a part of the Java language. Some of the reserve words are as follows (we'll cover these keywords throughout the book):

- `abstract`—It specifies that a class or method will be implemented later in a subclass
- `assert`—It signifies an assertion statement
- `boolean`—It is a data type that can hold True and False values only
- `break`—It is a control statement for breaking out of loops
- `byte`—It is a data type that can hold 8-bit data values
- `byvalue`—It is reserved for future use
- `case`—It is used in the `switch` statements to mark blocks of text
- `cast`— It is reserved for future use
- `catch`—It catches exceptions generated by the `try` statements
- `char`—It is a data type that can hold unsigned 16-bit Unicode characters
- `class`—It declares a new class
- `const`—It is reserved for future use
- `continue`—It sends control back outside a loop
- `default`—It specifies the default block of code in a `switch` statement
- `do`—It starts a `do-while` loop
- `double`—It is a data type that can hold 64-bit floating-point numbers
- `else`—It indicates alternative branches in an `if` statement
- `enum` —It declares enumerated type variables
- `extends`—It indicates that a class is derived from another class or an interface is derived from another interface
- `false` —It indicates one of the two permissible values for a `boolean` variable
- `final`—It indicates that a variable holds a constant value or that a method will not be overridden
- `finally`—It indicates a block of code in a `try-catch` structure that will always be executed
- `float`—It is a data type that holds a 32-bit floating-point number
- `for`—It is used to start a `for` loop
- `future`—It is reserved for future use
- `generic`—It is reserved for future use
- `goto`—It is reserved for future use

- `if`—It tests a true/false expression and branches accordingly
- `implements`—It specifies that a class implements an interface
- `import`—It references other classes
- `inner`—It is reserved for future use
- `instanceof`—It indicates whether an object is an instance of a specific class or implements a specific interface
- `int`—It is a data type that can hold a 32-bit signed integer
- `interface`—It declares an interface
- `long`—It is a data type that holds a 64-bit integer
- `native`—It specifies that a method is implemented with native (platform-specific) code
- `new`—It creates new objects
- `null`—It indicates that a reference does not refer to anything
- `operator`—It is reserved for future use
- `outer`—It is reserved for future use
- `package`—It declares a Java package
- `private`—It is an access specifier indicating that a method or variable may be accessed only in the class it's declared in
- `protected`—It is an access specifier indicating that a method or variable may only be accessed in the class it has declared (or a subclass of the class it has declared in or other classes in the same package)
- `public`—It is an access specifier used for classes, interfaces, methods, and variables indicating that an item is accessible throughout the application (or where the class that defines it is accessible)
- `rest`— It is reserved for future use
- `return`—It sends control and possibly a return value back from a called method
- `short`—It is a data type that can hold a 16-bit integer
- `static`—It indicates that a variable or method is a class method (rather than being limited to one particular object)
- `strictfp`—It restricts the precision and rounding of floating-point numbers
- `super`—It refers to a class's base class (used in a method or class constructor)
- `switch`—It is a statement that executes code on the basis of a test value
- `synchronized`—It specifies critical sections or methods in a multithreaded code
- `this`—It refers to the current object in a method or constructor
- `throw`—It creates an exception
- `throws`—It indicates what exceptions may be thrown by a method
- `transient`—It specifies that a variable is not a part of an object's persistent state
- `try`—It starts a block of code that will be tested for exceptions
- `true` —It indicates one of the two permissible values for a `boolean` variable
- `var`—It is reserved for future use
- `void`—It specifies that a method does not have a return value
- `volatile`—It indicates that a variable may change asynchronously
- `while`—It starts a `while` loop

Creating an Application

The Big Boss arrives and says, "So now you can write Java! Give me a demonstration!" You turn to your terminal and immediately your mind goes blank. What will you write?

Here's a sample Java application that we'll develop over the next few sections, all the way through the compiling and running stages. Place this code in a file named `App.java`:

```
public class App {
        public static void main(String[] args) {
   System.out.println("Hello from Java!"); }
}
```

If you are new to Java, this might look strange to you. The idea here is that this application will print the text "Hello from Java!" when you compile and run it. For example, here's how it looks in a DOS window under Windows:

```
c:\>java App
Hello from Java!
```

Not the most significant of programs but a good one to get us started. Let's take this program apart line by line.

Line 1—public class App

Here's the first line in `App.java`:

```
public class App { . . . }
```

This line indicates that we are creating a new Java class named `App`. After we translate this class into bytecodes, the JVM will be able to create objects of this class and run them. You'll learn all about classes in depth in Chapter 4; this code is just to get us started with Java programming.

Note the keyword `public` in the preceding code. This keyword is an *access specifier*, which you'll learn more about in Chapters 4 and 5. The `public` access specifier indicates that this class is available anywhere in a program that makes use of it.

Also note that if you make a class public, Java insists that you name the file after it. That is, you can only have one public class in a `.java` file. The reason for this is that the Java compiler will translate the `.java` file into a bytecode file with the extension "`.class`," which means that `App.java` will be translated into `App.class`; and if the JVM needs the `App` class, it'll know to look in the `App.class` file. Because JVM uses the name of the file to determine what public classes are in the file, you can only have one public class in a file. For that reason, the code for the `App` class must be in a file named `App.java` (note that Java is pretty particular about this, and capitalization counts here, even in Windows that normally doesn't pay any attention to capitalization). The actual implementation of the class we are defining here will go between the curly braces:

```
public class App { . . . }
```

Java always encloses blocks of code within curly braces, that is, "{" and "}".You'll see in Chapter 4 that the code inside the block has its own scope (its visibility to the rest of the program). Right now, let's continue building our application by continuing with the next line of the code.

Line 2—public static void main(String[] args)

Here's the next line of code in our application:

```
public class App { public static void main(String[] args) { . . . } }
```

What's happening here is that we are creating a *method* in the `App` class. A method in OOP is like a function or subroutine in standard programming—it's a block of code that you can pass control to and that can return a value. Methods provide handy ways of wrapping code into a single functional unit; when you call a method, JVM executes the code in the method.

You'll be introduced to methods formally in Chapter 4, but here, the idea is that we are creating a method named `main`, which is the method that the JVM will look for when it starts an application (applets do not have a `main` method). When it finds the `main` method, the JVM passes control to it, and we'll place the code we want to execute in this method's code block.

There are a few things to note before continuing. The `main` method must be declared with the `public` access specifier, which means it may be called outside its class. It must be declared `static` as well, which means that

`main` is a class method, not an object method (you'll see in Chapter 4). It must not return a value when it's finished executing, which is why we use the keyword `void` in this code (in other words, a return value of type `void` means that there actually is no return value). Finally, note the argument in the parentheses following the word `main: String[] args`. You place an argument list in the parentheses of a method declaration like this to indicate what values are passed to the method and can be used by the code in the method. In this case, we are indicating that `main` has passed an array of string values called `args`. These string values hold the values passed from the command line when you start the application; for example, if you type `java app Hello there`, "Hello" and "there" would be the two strings in the args array. The full details appear in Chapter 4. Because we won't use any command-line arguments in this application, we won't use `args` in the code for the `main` method.

This line of code then starts the `main` method. The whole job of this method is to print out the text "Hello from Java!" which is done in the next line of code.

Line 3—System.out.println("Hello from Java!");

The main method has one line of code in it:

```
        System.out.println("Hello from Java!"); }
    }
```

This is the actual line of code that does the work of printing the statement. In this case, we are using some of the code that the programmers at Sun have already created to display the text "Hello from Java!" In particular, the `java.lang` package's `System` class is used here. Libraries of classes are called *packages* in Java, and the `java.lang` package is built into every Java program, which means you don't have to take special steps to make use of it, as you do with other Java packages. The `java.lang` package's `System` class includes a *field* (that is, a data member of the class) called `out`, and this field in turn has a method named `println`, which does the actual displaying of text.

To refer to the `System` class's `out` field, we use the terminology `System.out`. To use the `out` field's `println` method (which stands for *print line*), we use the terminology `System.out.println`. To print the text "Hello from Java!" we pass that text to `System.out.println` by enclosing it in quotes.

Also note that this line of code ends with a semicolon. This end-of-statement convention is something that Java has inherited from C and C++ (in fact, Java has inherited a lot from C and C++), and you end nearly all statements in Java with a semicolon. If this isn't something you are used to, you'll pick it up pretty quickly because the Java compiler refuses to translate your code into bytecodes until the semicolons are in place.

That's it, then—you've created your new application and stored it in a file named `App.java`. What's the next step? To get it actually run, take a look at the next solution.

Compiling Code

The Big Boss is chomping a cigar while standing right behind you as you enter your new Java application into a file. "Hmm," says the Big Boss, clearly not impressed. "What's next?" "Now," you say, "we have to compile the program and then we can run it." "OK," the Big Boss says. "Amaze me."

To translate a Java program into a bytecode file that the JVM can use, you use the Java compiler `javac` (for example, on Windows machines, this program will be called `javac.exe`, which is in the Java bin directory). Here's how you use `javac` in general:

```
javac [options] [sourcefiles] [@files]
```

The arguments to `javac` are:

- `options`—Command-line options
- `sourcefiles`—One or more source files to be compiled (such as `App.java`)
- `@files`—One or more files that list source files

To compile `App.java`, use the following command:

```
C:\>javac App.java
```

Note that to make sure `javac` can find `App.java`, you should first make sure the current directory is the one that has `App.java` in it. For example, in Windows, if you've stored `App.java` in `C:\Nancy`, to compile

App.java, first make sure C:\Nancy is the current directory (you can use the change directory command, cd, like this: cd C:\Nancy). Then, run javac like this: C:\Nancy>javac App.java. Alternatively, you can specify the path of App.java like this: C:\>javac C:\Nancy\App.java.

In addition, note that in this book, we'll use the generic prompt C:\> for the command line, but this actually stands for the current working directory, whatever that may be in your case—for example, C:\Bertie\Development>, C:\Programs>, Unix prompts, such as %, /home/Nancy/programs:, and so on. That is to say C:\> is just going to be a placeholder for the actual prompt on your machine for the current directory you are in.

The Java compiler javac takes the file App.java (assuming there are no errors), compiles it, translates it, and then creates a new file called App.class. If errors occur, the Java compiler will tell you what they are, including what line of code is wrong—in this case, we have forgotten the println method and tried to use the one called printline:

```
C:\>javac App.java
App.java:3: error: cannot find symbol
     System.out.printline("Hello from Java!"); }
                ^
  symbol:    method printline(String)
  location:  variable out of type PrintStream
1 error
```

When App.java is successfully compiled to bytecode, the new file, App.class, contains all that JVM will need to create objects from the App class. So, we have created App.class. Now, how do you actually run it in JVM? See the next solution.

Compiling Code: Using Command-Line Options

"Hmm," says the NP, "I have got a problem. I like to keep all my .class files in the same directory, but sometimes, I forget to copy the new versions of those files to that directory." You say, "There's a compiler option that's perfect for you. It's called the -d option. Using that option, you can have the compiler place bytecode files into any target directory you want." "Well," says the NP. "Now, if I can only remember to use that option…"

There are quite a number of options—that is, command-line directives—you can use with javac. For example, here's how you can use the -d option to have javac place the file App.class in an existing directory called temp, which, in this case, is a sub-directory of the current directory:

```
javac -d temp App.java
```

The following is the list of javac options; note that the options that start with -X (called *non-standard options*) are marked that way by Sun because they may change in the future:

- -classpath classpath—It sets the user class path, overriding the user class path in the CLASSPATH environment variable. If neither CLASSPATH nor -classpath is specified, the user class path will be the current directory. Note that if the -sourcepath option is not used, the user class path is searched for source files as well as class files.
- -d directory—It sets the destination directory for .class files. For readers who know what Java packages are, if a class is a part of a package, javac puts the .class file in a sub-directory that reflects the package name, creating directories as needed. For example, if you specify -d c:\classes and the class is called com.package1.Class1, the class file is called c:\classes\com\package1\Class1.class. If -d is not specified, javac puts the class file in the same directory as the source file. Note that the directory specified by -d is not automatically added to your user class path.
- -deprecation—It shows a description of each use or override of a deprecated member or class. (Without -deprecation, javac only shows the names of source files that use or override deprecated members or classes.)
- -encoding—It sets the source file encoding name. If -encoding is not specified, the platform default converter is used.

- -g—It generates all debugging information, including local variables. By default, only line number and source file information are generated.
- -g:none—It makes the compiler unable to generate any debugging information.
- -g:{keyword list}— It generates only some kinds of debugging information, specified by a comma-separated list of keywords. The valid keywords are source (source file debugging information), lines (line number debugging information), and vars (local variable debugging information).
- -nowarn— It disables all warning messages.
- -O—It optimizes code for performance in terms of the quickest execution time. Note that using the -O option may slow down compilation, produce larger class files, and make the program difficult to debug. Note that before version 1.2, the -g and -O options of javac could not be used together. As of version 1.2, you could combine the -g and -O options, but you might get odd results, such as missing variables and relocated or missing code. The -O option no longer automatically turns on -depend or turns off -g.
- -sourcepath—It specifies the source code path to search for class or interface definitions. As with the user class path, source path entries are separated by semicolons (;) and can be directories, .jar (Java Archive) files or zip files. If you use packages, the local pathname within the directory or archive must reflect the package name, as you'll see later. Note that classes found through the class path are subject to automatic recompilation if their source code files are found.
- -verbose—It creates "verbose" output. This includes information about each class loaded and each source file compiled.
- -X—It displays information about non-standard options and quits.
- -Xdepend—It searches all reachable classes for more recent source files to recompile. This option will more reliably discover classes that need to be recompiled, but it can slow down the compilation process dramatically.
- -Xstdout—It sends compiler messages to System.out. By default, the compiler messages go to System.err, which you'll learn more about later.
- -Xverbosepath—It describes how paths and standard extensions were searched to find source and class files.
- -Joption—You use this option to pass an option to the Java launcher called by javac. For example, -J-Xms64m sets the start-up memory to 64 MB. Although this option does not begin with -X, it's not a standard option of javac. It's a common convention for -J to pass options to the underlying JVM executing applications written in Java.

Cross-Compilation Options

Cross-compilation options are considered an advanced topic; javac supports cross-compiling, where classes are compiled with the bootstrap (default) and extension classes of a different Java platform implementation. You must use -bootclasspath and -extdirs when cross-compiling. Here are the cross-compilation options:

- -target version—It generates class files that will work on JVMs with the specified version.
- -bootclasspath bootclasspath—It cross-compiles against the specified set of boot classes. As with the user class path, boot class path entries are separated by semicolons (;) and can be directories, .jar files or .zip files.
- -extdirs directories—It cross-compiles against the specified extension directories; directories is a semicolon-separated list of directories. Each .jar file in the specified directories is automatically searched for class files.

Compiling Code: Checking for Deprecated Methods

"Good Lord!" says the NP, "How can I keep all the deprecated methods in Java straight? Now that I'm upgrading to Java SE 8, I don't know what's obsolete and what's not!" "That's an easy one", you say. "The Java compiler `javac` will now tell you if you are using a method that's been deprecated, which it does not use unless you specifically asked it to. Even better, you can use the `-deprecation` option to make sure you get all the details

The `-deprecation` option is a good one, and it's the standard option we use to make sure we avoid deprecated methods. Suppose you have a program that uses `addItem` (a deprecated method) of the `List` class, and when you try to compile it, `javac` gives you the following result:

```
C:\>javac App.java
Note: App.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

That's not much help. However, using the `-deprecation` option, you can pinpoint the exact problem:

```
C:\>javac -deprecation App.java

App.java:7: warning: [deprecation] addItem(String) in List has been deprecated
          list.addItem("Kogent");
              ^
1 warning
```

NOTE

You can also use –Xlint compiler option to pinpoint the warning point in the program.

Running Code

The Big Boss is getting impatient. You've written a new application and compiled it without errors for the first time (which you can feel proud of), but nothing has really happened that the BB can see. It's time to run the new application.

You run Java applications with the program named Java (in Windows, for example, this is the `java.exe` file in the Java `bin` file). The Java program, called the `java` tool, is what actually runs JVM. Here's how you can use the `java` tool:

```
java [options] class [argument ...]
java [options] -jar file.jar [argument ...]
```

The parameters used in the preceding lines are explained as follows:

- `options`—Command-line options, which we'll cover in a topic coming right up.
- `class`—The name of the class to be invoked.
- `file.jar`—The name of the Java Archive (JAR) file to be invoked. This is used only with `-jar`. JAR files are covered in Chapter 27.
- `argument`—A command-line argument passed to the `main` method.

For example, to run the application named app, which is in the file `App.class`, you could execute the following command at the command line (note that you omit the ".class" part of `App.class` here):

```
C:\>java App
```

The result appears immediately:

```
C:\>java App
Hello from Java!
```

TIP

To make sure the java tool can find App.class in this example, you should first make sure the current directory is the one that has App.class in it. For example, in Windows, if you've stored App.class in C:\Nancy, to run it, first make sure C:\Nancy is the current directory (you can use the change directory command, cd, like this: cd C:\Nancy). Then, run Java like this: C:\Nancy>java App. Alternatively, you can specify the path of App.class like this: C:\>java –classpath C:\Nancy App.

You can see how this works in a DOS window under Windows in Figure 1.3:

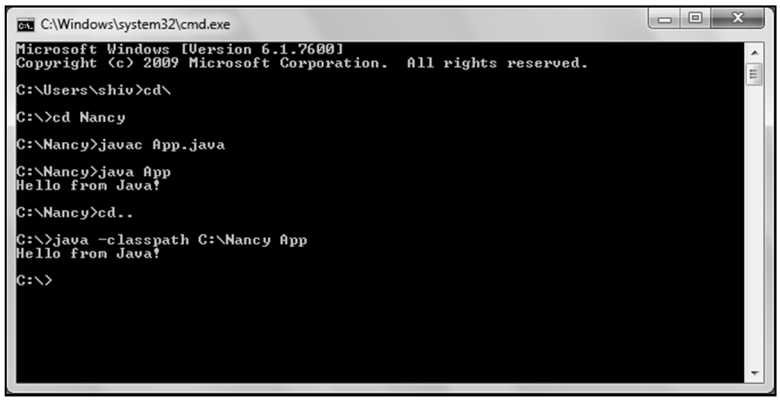


Figure 1.3: Running an Application in a DOS Window

That's all it takes—now you've written, compiled, and run your first application. Congratulations!

Note that if your application isn't responding or if you want to stop it for some reason, you can type `Ctrl+C`. If that doesn't work, try the `Esc` key.

You'll also see how to create windowed applications in this book, and when you run one of these applications with the java tool, you get the results as shown in Figure 1.4 below.

There's one thing to note about Figure 1.4—the console window (a DOS window here) hangs around in the background and waits for the application to finish before continuing (that is, before the DOS prompt reappears in this case). If you don't want a console window associated with your windowed application, you can use the `javaw` tool like this:

```
C:\>javaw App
```

Here's how you use `javaw` in general—just like the `java` tool:

```
javaw [options] class [argument ...]
javaw [options] -jar file.jar [argument ...]
```

The parameters used by `javaw` are as follows:

- `options`—Command-line options, which we'll cover later in the chapter.
- `class`—The name of the class to be invoked.
- `file.jar`—The name of the Java Archive (JAR) files to be invoked. This is used only with `-jar`. JAR files are covered in Chapter 27.
- `argument`—A command-line argument passed to the `main` method.



Figure 1.4: Running a Windowed Application

When you launch a Java windowed application like this, the console window does not wait for the application to finish; if you are running in DOS, the windowed application appears and the DOS prompt reappears in the DOS window. This gives a more professional feel to those applications that you release for general use.

While we are discussing compiling and running code, there's another detail we should cover—the command-line options you can use with the `javac` and `java` commands. We'll take a look at them in the next two topics.

Running Code: Using Command-Line Options

"Well," says the NP, "I have another problem. I have stored all my `.class` files in one directory, but I don't want to keep switching to that directory to run them." "Another easy problem to solve," you say. "You can use the `java` tool's `-classpath` option or set the `CLASSPATH` environment variable so that the Java compiler will search for your classes correctly"

TIP

For more on CLASSPATH, an important topic in Java programming, see the upcoming solution "Finding Java Classes with CLASSPATH".

We'll take a look at using the command-line options in this solution; you use these options with the `java` and `javaw` tools as follows (for a discussion of these tools, see the solution "Running Code," earlier in this chapter):

```
java [options] class [argument ...]
java [options] -jar file.jar [argument ...]
javaw [options] class [argument ...]
javaw [options] -jar file.jar [argument ...]
```

The command-line options you can use with these tools are as follows (note that non-standard options, which means they might not be supported in the future, begin with an `X`):

- `-classpath classpath` or `-cp classpath`—It specifies a list of directories, `.jar` files or `.zip` files to search for class files. You separate class path entries with semicolons (;). Note that specifying **-**`classpath` or **-**`cp` overrides any setting of the `CLASSPATH` environment variable. Used with `java` or `javaw`, **-**`classpath` or **-**`cp` only specifies the class path for user classes. If **-**`classpath` and **-**`cp` are not used and `CLASSPATH` is not set, the user class path is limited to the current directory, which is referred to with a dot (.). See the topic "Finding Java Classes with `CLASSPATH`" later in this chapter for more information.
- `-Dproperty=value`—It sets a system property value.
- **-**`jar`—It executes a program encapsulated in a JAR file. The first argument is the name of a JAR file instead of a start-up class name. When you use this option, the JAR file is the source of all user classes, and the other user class path settings are ignored.
- **-**`verbose` *or* **-**`verbose:class`—It displays information about each class loaded.
- **-**`verbose:gc`—It reports on each garbage collection event. Garbage collection involves automatic memory management in Java.
- **-**`verbose:jni`—It reports information about the use of native (that is, platform-specific) methods and other Java Native Interface activity.
- **-**`version`—It displays version information and exits.
- `-?` or **-**`help`—It displays usage information and exits.
- `-X`—It displays information about non-standard options and exits.
- `-Xbootclasspath:bootclasspath`—It specifies a semicolon-separated list of directories, `.jar` files or `.zip` files to search for boot class files. Note that these will be used in place of the boot class files included with Java itself.
- **-**`Xdebug`—It starts with the debugger enabled.
- `-Xnoclassgc`—It disables class garbage collection.

- `-Xmsn`—It indicates the initial size of the memory pool you want to use (this value must be greater than 1000). To multiply the value by 1000, append the letter k. To multiply the value by 1 million, append the letter m. The default value is `1m`.
- `-Xmxn`—It specifies the maximum size of the memory pool (this value must be greater than 1000). To multiply the value by 1000, append the letter k. To multiply the value by 1 million, append the letter m. The default value is `64m`.
- `-Xrunhprof[:help][:<suboption>=<value>,...]`—It enables CPU, heap, or monitor profiling. This option is usually followed by a list of comma-separated pairs of the form `<suboption>=<value>`.
- `-Xrs`—It reduces the use of operating system signals.
- `-Xcheck:jni`—It performs additional checks for Java Native Interface (JNI) functions.
- `-Xfuture`—It performs strict class-file format checks.

Commenting Your Code

The Programming Correctness Czar (PCC) comes in and looks at you reprovingly. "What's wrong, PCC?" you ask. "It's your code," the PCC says. "I can't make heads or tails of what's going on in it." "I guess I forgot to comment it," you say. "I guess you did," the PCC says. "Fix it."

Sometimes, a code can be very cryptic and hard to decipher. For that reason, Java lets you place descriptive comments in your code to let you explain to anyone who reads that code how the program works and what it does. As an example, let's add comments to the application we have already developed in the previous topics:

```
public class App { public static void main(String[] args) {
   System.out.println("Hello from Java!"); }
}
```

Java supports three types of comment, two of which are taken from C++. You can surround a comment of any length with the characters "**/***" and "***/**" as follows:

```
/* This application prints out "Hello from Java!" */
public class App { public static void main(String[] args) {
   System.out.println("Hello from Java!"); }
}
```

The Java compiler will ignore all the text between the `/*` and `*/` markers. You can split comments between these markers across multiple lines as follows:

```
/* This application prints out "Hello from Java!"
Created by: G. Whiz, 1/1/00 */
public class App {
        public static void main(String[] args) {
   System.out.println("Hello from Java!"); }
}
```

In fact, in many corporate environments, you are expected to use a standard comment header, created with the `/*` and `*/` forms of comment, for all new code. It might look something as follows:

```
/*******************************************************
* This application prints out "Hello from Java!" *
* *
* Author: G. Whiz *
* Imports: None *
* Parameters: Command-line arguments *
* Returns: None *
* Assumptions: None *
* Creation date: 1/1/00 *
* Last Update: 1/1/01 *
*******************************************************/
public class App {
        public static void main(String[] args) {
   System.out.println("Hello from Java!"); }
}
```

Java also supports a one-line comment using a double slash (//). The Java compiler will ignore everything on a line after the // marker, so you can create whole lines that are comments or just add a comment to an individual line as follows:

```
/* This application prints out "Hello from Java!" */
public class App { //Create the App class
   //Create main(), the entry point for the application.
   public static void main(String[] args) {
   //Print out the message with
   System.out.println("Hello from Java!"); }
}
```

Finally, Java also supports a documentation comment, which starts with "/**" and ends with "*/."This comment is designed to be used with the `javadoc` tool, which can create documentation for you nearly automatically. An example of using /** and */ is as follows:

```
/** This application prints out "Hello from Java!" */
public class App {
        public static void main(String[] args) {
   System.out.println("Hello from Java!"); }
}
```

Commenting your code can be invaluable in team environments where you share your code source files with others. It's also handy if someone else is going to take over a project that you have been working on.

Import Java Packages and Classes

"Hmm," says the NP. "I have got a problem. The Design Team Coordinator told me to use the `Date` class for printingout the current date in my application, but Java doesn't seem to have ever heard of the `Date` class—I get an error every time I try to use it." "That's because the `Date` class is part of the `java.util` package, and you have to import that package before you can use it." "Import it?" the NP asks.

The classes that Sun has created for you to use are stored in class libraries called packages. To make a class in a package available to your code, you have to import the package, which means the compiler will search that package for classes. You can also import individual classes that are not part of a package. By default, only the basic Java statements are available to you in an application, that is, the ones in the core `java.lang` package. The compiler automatically imports the `java.lang` package for you, but to use the rest of the classes that come with Java, you'll have to do your own importing with the `import` statement. Here's how you use that statement:

```
import [package1[.package2...].](classname|*);
```

Note that you put a dot (.) between the package and the class names to keep them separate. The standard Java packages themselves are stored in a large package called `java`, so the `util` package is really called the `java.util` package (there are other large packages like the `java` package available; for example, the extensive swing package is stored in the `javax` package). You can refer to the `Date` class in `java.util` as `java.util.Date`. Here's how to import that class into a program:

```
import java.util.Date;
public class App { . . .
}
```

Note that if you are going to use `import` statements for importing classes into a program, the `import` statements should be at the top of the code. Now, we are free to use the `Date` class as follows (note that we are creating an object from the `Date` class by using the Java `new` operator, which you'll learn more about in Chapter 4):

```
import java.util.Date;
public class App {
        public static void main(String[] args) {
   System.out.println("Today = " + new Date()); }
}
```

When you run this application, you'll see the current date displayed as follows:

```
C:\>java App
Today = Tue July 17 14:05:15 IST 2012
```

You can see that by studying the general form of the preceding `import` statement, there's also a shorthand technique that loads in all the classes in a package—you can use an asterisk (*) as a wildcard to stand for all the

classes in a particular package. Here's how that would look if you wanted to import all the classes in the `java.util` package at once:

```
import java.util.*;
public class App {
        public static void main(String[] args) {
   System.out.println("Today = " + new Date()); }
}
```

TIP

Importing packages and classes only indicates to the compiler where to look for the code it needs—it does not increase the size of your code. For that reason, the bytecode file App.class will be of the same size regardless of whether you use the import java.util.Date; statement or the import java.util.; statement.*

This is fine if you stick with importing the Sun-provided classes because Java knows where to look for the classes it was installed with. But what if you want to import your own classes or ones provided by a third party?

Here's an example. Suppose you have a user-defined package named `printer`, having a class named `Printer` in a file named `Printer.java`, and that class has one method named `print`:

```
package printer;
public class Printer {
        public void print() { System.out.println("Hello from Java!"); }
}
```

You might want to make use of the `print` method in other classes. In this case, we are creating a new object of the `Printer` class using the `new` operator and using that object's `print` method in an application named `App`:

```
public class App {
        public static void main(String[] args) { (new Printer()).print(); }
}
```

To do this, you can import the `Printer` class as follows (note that you can also place the code for the `Printer` class in the same file as the `App` class, in which case you wouldn't have to import the `Printer` class):

```
import printer.*;
public class App {
        public static void main(String[] args) { (new Printer()).print(); }
}
```

Here, we have learned how to import a Java built-in and a user-created class in our Java program. To learn about the implementation part and how to actually create and import packages in our classes, refer Chapter 27.

However, suppose you want to store all your classes in a directory named, say, c:\classes. How will the Java compiler find `Printer.class` there? To answer that question, take a look at the next solution on `CLASSPATH`.

Finding Java Classes with **CLASSPATH**

"That Johnson!" the NP says. "He gave me a new Java class file, `johnson.class`, to work with, and it's supposed to solve my problems with that spreadsheet. But, Java claims it can't find `johnson.class`!" "Where are you keeping that file?" you ask. "In a special directory I made for it," the NP says, "called `darnjohnson`." "That's your problem," you say. "You have to include the `darnjohnson` directory in your class path."

By default, Java will be able to find its bootstrap classes (the ones it comes with), extension classes (those that use the Java Extension Framework), and classes in the current directory (that is, where you are compiling your program). Classes can be stored in the `.class` files, `.jar` files, and `.zip` files. Java can search all these types of files.

NOTE

Bootstrap classes are the classes that come by default with JDK, and Extension classes are the classes that use the Java Extension Framework. But, what if you want to have Java search for classes in another directory or in a .jar file supplied by a third party? You can do that with the CLASSPATH environment variable because Java uses this variable to determine where you want to search for classes.

Here's an example that was first introduced in the previous solution. Say that you have a class named `Printer` in a file named `Printer.java` and that class has one method named `print`:

```
public class Printer {
   public void print() { System.out.println("Hello from Java!"); }
}
```

In the previous solution, you want to use the `print` method in another class—as in this case where we are creating a new object of the `Printer` class by using the new operator and using that object's `print` method in an application named `App`:

```
import printer.*;
public class App {
        public static void main(String[] args) { (new Printer()).print(); }
}
```

This works if `Printer.class` is in the same directory in which you are compiling this application, because the Java compiler will search the current directory by default. But, suppose you want to store all your classes in a directory named `c:\classes`. How will the Java compiler find `Printer.class` there?

To make the Java compiler search `c:\classes`, you can set the `CLASSPATH` environment variable to include that directory. By default, there are no paths or directories in `CLASSPATH`, but you can add a semicolon-separated list to `CLASSPATH`, like this (note that it's important here not to have any spaces around the equals sign):

```
SET CLASSPATH=c:\classes;c:\newclasses
```

You can also determine the current `CLASSPATH` setting by typing the `SET CLASSPATH` at the command line and pressing Enter. To permanently set the classpath in Windows 7, follow these steps:

1. Click the Windows icon on the taskbar, right-click the Computer, and select the Properties option. You can also reach the same location by opening the Control Panel | System and Security | System.
2. Click the Advanced system settings option in the System Properties window. This opens up the System Properties dialog box.
3. Click the Environment Variables button to open the Environment Variables dialog box.
4. Edit the `CLASSPATH` setting the way you want.
5. You may append the new classpath to the already existing `CLASSPATH` variable or create a new `CLASSPATH` variable.
6. Finally, click OK.

Note that if you are appending the CLASSPATH variable, the different paths must be separated by semicolons (;).

Now, the Java compiler (and other Java tools, such as the `java` tool) will know enough to search `c:\classes` and `c:\newclasses` automatically. This means that the following code will now work if `Printer.class` is in `c:\classes` because that directory is in `CLASSPATH`:

```
import printer.*;
public class App {
        public static void main(String[] args) { (new Printer()).print(); }
}
```

You can append the current settings in `CLASSPATH` to a new setting, like this:

```
SET CLASSPATH=c:\classes;c:\newclasses;%CLASSPATH%
```

Note that you can also search `.jar` and `.zip` files for classes, as shown here:

```
SET CLASSPATH=server.jar;classes.zip;%CLASSPATH%
```

Originally, `CLASSPATH` was a big headache for beginners in Java programming because no classes were considered bootstrap classes, which meant that you had to set up and understand `CLASSPATH` before you could use Java at all. That's been fixed with the concept of bootstrap classes, which are the classes that come with Java (and are searched automatically). However, if you want to use non-standard packages or store your own classes in other directories, it's important to know how to set `CLASSPATH`.

Summary

In this chapter, you got familiar with Java 8, its environment, and designing of programs. The chapter introduced you to Swings and Utilities. In addition, it discussed the enhancements in JDBC and provided a brief description of garbage collector. You also learned how to install Java. Toward the end, you developed simple Java programs with the introduction of how to import the packages.

In the next chapter, you will learn about the variables, arrays, and strings.



2

Variables, Arrays, and Strings

If you need an immediate solution to:	*See page:*
What Data Types are Available?	37
Creating Integer Literals	37
Creating Floating-Point Literals	38
Creating Boolean Literals	39
Creating Character Literals	39
Creating String Literals	40
Creating Binary Literals	40
Using Underscores in Numeric Literals	41
Declaring Integer Variables	42
Declaring Floating-Point Variables	42
Declaring Character Variables	43
Declaring Boolean Variables	43
Initializing Variables Dynamically	44
Conversion between Data Types	45
Declaring One-Dimensional Arrays	47
Creating One-Dimensional Arrays	47
Initializing One-Dimensional Arrays	48
Declaring Multi-Dimensional Arrays	48
Creating Multi-Dimensional Arrays	48
Initializing Multi-Dimensional Arrays	49
Creating Irregular Multi-Dimensional Arrays	50
Getting an the Length of an Array	51

If you need an immediate solution to:	*See page:*
Understanding General Form of Static Import	51
Importing Static Members	51
The String Class	52
Creating Strings	56
Getting String Length	57
Concatenating Strings	58
Getting Characters and Substrings	58
Searching For and Replacing Strings	59
Changing Case in Strings	60
Checking for Empty String	60
Formatting Numbers in Strings	60
The StringBuffer Class	61
Creating StringBuffers	63
Getting and Setting StringBuffer Lengths and Capacities	64
Setting Characters in String Buffers	64
Appending and Inserting Using StringBuffers	65
Deleting Text in StringBuffers	65
Replacing Text in StringBuffer	65
Using the Wrapper Class	66
Autoboxing and Unboxing of Primitive Types	67
Learning the Fundamentals of Varargs Methods	69

In Depth

This chapter begins our discussion of *Java syntax*, and you'll see a great deal of syntax usage in this chapter. We are going to cover how Java stores and retrieves data in variables, arrays, and strings. Working with data is a fundamental part of any significant program, and the information in this chapter is essential for any Java programmer. Even if you are familiar with Java and have already been programming in it, take a look at the material in this chapter because there's a lot coming up.

Variables

There are various kinds of variable that serve as containers for storing data in the memory. Different kinds of variable store data in different formats, and the selection of variable should be made depending upon the data that they will be storing and the memory that will be required to be kept aside for storing it. For example, an integer variable type, the `int` type, can store up to 4 bytes (or 32 bits) and contain only integer values between 2,147,483,648 and 2,147,483,647. There are some more variables that Java provides such as floating-point numbers, short, long, byte, and char, and all of them will be explained in this chapter.

Before you use a variable in Java, you must *declare* it, specifying its data type. Here's how you declare variables in Java:

```
type name [= value][, name [= value]...];
```

Here's an example showing how to declare a variable of the `int` type, which means a numeric value will be stored in it (the variable is named `days`):

```
public class App {
        public static void main(String[] args) {
  int days;
  . . . }
}
```

This code allocates 32 bits of storage in memory and labels the location of that storage, as far as the Java compiler is concerned, as `days`, which means you can now refer to that name in code. Here's how to store a numeric value 365 in the `days` variable using the Java assignment operator (=):

```
public class App {
        public static void main(String[] args) {
  int days;
  days = 365;
  . . . }
}
```

Here, the value 365 is an integer *literal*, which means a literal value that you place directly in your code. We'll take a look at what kinds of literal Java allows throughout this chapter. To verify that `days` now hold 365, you can print it out on the console:

```
public class App {
        public static void main(String[] args) {
  int days;
  days = 365;
  System.out.println("Number of days = " + days); }
}
```

Here's the result of this code:

```
C:\>java App
Number of days = 365
```

As you can see, we have created a variable, stored data in it, and fetched the same data to print it on the screen. That's how it works.

There's also a convenient shortcut that lets you initialize a variable when you declare it. Here, `days` are declared and initialized to 365 in one step:

```
public class App {
        public static void main(String[] args) {
  int days = 365;
```

```
        System.out.println("Number of days = " + days); }
    }
```

The `int` type is only one kind of simple variable you can use. Mentioned below are the various possibilities:

- **Integers**—Refer to the types `byte, short, int`, and `long` that hold signed, whole-value numbers.
- **Floating-point numbers**—Refer to the types `float` and `double` that hold signed floating-point numbers.
- **Character**—Refers to the `char` type that holds characters such as letters and numbers.
- **Boolean**—Refers to the type that holds only two types of values: *true* and *false*.

We'll take a closer look at all these in the "Immediate Solutions" section, including what range of values each can hold. Together, these types make up what are called *simple data types* in Java. Each of these types represents a single data value, not a compound data value (as opposed to an array, which is also discussed in this chapter). You can store one data item in a variable made up of any simple data type, and that data item must fit in the range allowed for that data type.

Data Typing

Java puts considerable emphasis on its data types. It's a *strongly-typed* language, which means it insists that the simple variables you declare and use must fit in the listed types.

Every simple variable must have a type (and in fact, every expression—every combination of terms that Java can evaluate to get a value—has a type as well). Also, Java is very particular about maintaining the integrity of those types, especially if you try to assign a value of one type to a variable of another type. In fact, Java is more strongly typed than a language such as C++. In C++, for example, you can assign a floating-point number to an integer, and C++ will handle the type conversion for you, but you cannot do that in Java. You can, however, convert between certain data types in Java such as between the integer types. We'll take a look at that later in this chapter.

TIP

When working with variables, you might find the Java compiler issuing a lot of errors and warnings about data types, which can take some time getting used to; bear in mind that the inspiration for making Java very particular about adhering to data types and not mixing them easily is to prevent errors in your code.

That's an overview of what's going on in Java with simple data types and variables; it's now time to take a look at compound data storage in depth, which as far as this chapter is concerned means *arrays*.

Arrays

Simple types are good only for storing single data items. However, sometimes the data that needs to be stored is a lot more complex. For instance, you want to start a new bank, say, JP Bank (short for Java Programming Bank) and keep a track of the amount in each account indexed by account number. This situation presents the need to work with compound data. The usage of arrays is appropriate in such a scenario.

An array allows grouping of simple data to form a *compound data structure* that can be referred to by using a single name. Every data item stored in that compound data structure is stored at a particular position in the array and can be referred by the numeric index (or position) of the data in the data structure. It is vital to store the data in indexed form with the index sorted numerically because computers perform millions of operations in a second and data stored in indexed form can be referenced using the numerical value. This way the entire data structure can be traversed by simply incrementing the index's value, and the values can be accessed with ease.

Here's an example. In this case, we'll start the Java Programming Bank out with 100 new accounts, and each one will have its own entry in an array named `accounts[]`. The square braces at the end of the `accounts[]` represent that it is an array and the value of the index can be placed enclosed in those braces to refer to the value stored at a particular position. Here's how we create the `accounts[]` array, making each entry in it of the floating-point type `double` for extra precision. First, we declare the array; then, we create it with the `new` operator, which is what Java uses to actually allocate memory:

```
    public class App {
```

```
            public static void main(String[] args) {
      double accounts[];
      accounts = new double[100];
      . . .
```

Now that we have created an array with 100 items, we can refer to those items numerically, like this (note that we are storing $43.95 in account 3 and printing that amount out):

```
    public class App {
            public static void main(String[] args) {
      double accounts[];
      accounts = new double[100];
      accounts[2] = 43.95;
      System.out.println("Account 3 has $" + accounts[2]);    }
    }
```

Here's the result of this program:

```
    C:\>java App
    Account 3 has $43.95
```

As you can see, you can now refer to the items in the array by using a numeric index, which organizes them in an easy way. In Java, the lower bound of an array you declare this way is 0, so the statement `accounts = new double[100]` creates an array whose first item is `accounts[0]` and last item is `accounts[99]`.

You can combine the declaration and creation steps into one step, as shown below:

```
    public class App {
            public static void main(String[] args) {
      double accounts[] = new double[100];
      accounts[2] = 43.95;
      System.out.println("Account 3 has $" + accounts[2]); }
    }
```

You can also initialize an array with values when you declare it if you enclose the list of values you want to use in curly braces, as you'll see in this chapter. For example, this code creates four accounts and stores 43.95 in `accounts[3]`:

```
    public class App {
            public static void main(String[] args) {
      double accounts[] = { 0, 0, 0, 43.95 };
            System.out.println("Account 4 has $" + accounts[3]);      }
    }
```

It turns out that some of the customers in the Java Programming Bank are unhappy, though; they want a checking account in addition to a savings account. How will you handle this and still keep things indexed by account number?

The `accounts[]` array is a one-dimensional array, also called a *vector*, which means you can think of it as a single list of numbers that you can index with one number. However, arrays can have multiple dimensions in Java, which means you can have multiple array indexes. In this next example, we'll extend `accounts[]` into a two-dimensional array—`accounts[][]`—to handle both a savings account and a checking account. The first index of `accounts[][]` will be 0 for savings accounts and 1 for checking accounts, and the second index will be the account number as before. Here's how this works in code:

```
    public class App {
            public static void main(String[] args) {
      double accounts[][] = new double[2][100];
      accounts[0][2] = 43.95;
      accounts[1][2] = 2385489382.06;
      System.out.println("Savings account 3 has $" + accounts[0][2]);
      System.out.println("Checking account 3 has $" + accounts[1][2]); }
    }
```

After creating a two-dimensional array, values in it can be referenced using two index values. For example, here we are assuming the value `0` for savings account and `1` for checking account. The savings account details in the account number 3 can now be referenced as `accounts[0][2]` and that in the checking account can be referenced as `accounts[1][2]`. Shown here are the results when you run this application:

```
    C:\>java App
    Savings account 3 has $43.95
    Checking account 3 has $2.38548938206E9
```

Note that we have given account 3 a checking balance of $2,385,489,382.06 (wishful thinking) and that Java has printed that out as 2.38548938206E9. This is Java's shorthand for $2.38548938206X10^9$—not an inconsiderable bank balance by any means.

TIP

You'll see a lot of arrays in this chapter, but you should know that Java now supports much more complex data structures than arrays. These data structures are built into the language. Java now supports hashes and maps as well as other types of data structure, and as you'll see all that when we take a look at the collection classes in the forthcoming chapter.

Strings

You may have noticed that we have been using the + operator to create the text to print in the previous examples like this:

```
public class App {
        public static void main(String[] args) {
  double accounts[][] = new double[2][100];
  accounts[0][2] = 43.95;
  accounts[1][2] = 2385489382.06;
  System.out.println("Savings account 3 has $" + accounts[0][2]);
  System.out.println("Checking account 3 has $" + accounts[1][2]); }
}
```

That's because their own class in Java—the `String` class—supports text strings and you can think of the `String` class as defining a new data type.

For example, here's how we create a string named `greeting` that holds the text "Hello from Java!":

```
public class App {
        public static void main(String[] args) {
  String greeting = "Hello from Java!";
  . . .
```

Now we can treat this string as we would other types of variables, including printing it out:

```
public class App {
        public static void main(String[] args) {
  String greeting = "Hello from Java!";
  System.out.println(greeting); }
}
```

Here's the result of this application:

```
C:\>java App
Hello from Java!
```

Although strings are not one of the simple data types in Java, yet they deserve a place in this chapter because most programmers treat them as they would any other data type. In fact, many programmers would argue that strings should be a simple data type in Java as they are in other languages. The reason they are not has to do with Java lineage, which stretches back to C. C has no string simple data type; in C, you handle strings as one-dimensional arrays of characters, which is pretty awkward. One of the things that made programmers happy about C++ was that most implementations included a `String` class that you could use much as you would any other data type. Java follows this usage, implementing strings as a class, not as an intrinsic data type, but string handling is so fundamental to programming that it makes sense to start looking at string variables in this chapter.

There are two string classes in Java—`String` and `StringBuffer`. You use the `String` class to create text strings that cannot change, and you can use `StringBuffer` to create strings you can modify. As you can see in the preceding code, you can use strings much as you would any simple data type in Java. We'll take a look at using strings in this chapter as well as in the next chapter (which is on using operators such as + and –). We'll also take a look at using operators on strings.

That's enough overview for now—it's now time to start creating and using variables, arrays, and strings.

Immediate Solutions

What Data Types are Available?

"Say," the Big Boss (BB) says, "how about writing a Java program to manage the company's debt?" "We are in debt?" you ask. "Just a little," the BB replies. "How little?" you ask. "About $2,848,238,493, 902.77," says the BB. "Hmm!" you say. "Sounds like a job for floating-point numbers."

What simple data types can you use to create variables in Java? You'll find them in Table 2.1 below. The simple data types can be broken up by category, as we did at the beginning of this chapter, like this:

- **Integers**—Refer to the `byte`, `short`, `int`, and `long` types. They contain signed, whole-value numbers.
- **Floating point numbers**—Refer to the `float` and `double` types. They contain signed, floating-point numbers.
- **Character**—Refers to the `char` type that holds characters such as letters and numbers.
- **Boolean**—Refers to the type that holds only two types of values: true and false.

Table 2.1: Variable types

Variable type	Bytes of storage	Range
`boolean`	`1 bit`	True, False
`byte`	`1`	-128 to 127
`char`	`2`	'\u0000 to '\uffff
`double`	`8`	4.9E-324 to 1.7976931348623157E308
`float`	`4`	1.4E-45 to 3.4028235E+38
`int`	`4`	-2,147,483,648 to 2,147,483,647
`long`	`8`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`short`	`2`	-32,768 to 32,767

That's an overview of what simple data types are available; to put each of them to work, see the following solutions.

TIP

It's worth realizing that you can't completely count on the numeric precision listed in Table 2.1 because Java implementations do differ by machine. However, you can use the StrictMath class (you can find more about these classes at http://docs.oracle.com/javase/8/docs/api/) to ensure that the same mathematical precision is used no matter what platform your program is running on.

Creating Integer Literals

The Novice Programmer (NP) appears and says, "So, how do I assign a *hexadecimal* value—that is, base 16—to a variable in Java?" You say, "You have to use a hexadecimal literal, which starts with the characters *0x* or *0X*."

A literal is a constant value that you can use directly in your Java code, and there are a number of rules that govern them. We have already used integer literals in this chapter, which is the most common type programmers use. Here's the example:

```
public class App {
        public static void main(String[] args) {
int days = 365;
System.out.println("Number of days = " + days); }
}
```

Here, we are assigning an integer literal with a value of 365 to the variable `days`. By default, integer literals are of the `int` type. However, if you assign them to other integer types, such as `short`, Java converts the literal type automatically. On the other hand, `long` values can have more digits than `int` values, so Java provides an explicit way of creating `long` literals: You append an *L* to the end of the literal. Here's an example:

```
public class App {
        public static void main(String[] args) {
long value;
value = 1234567890123456789L;
System.out.println("The value = " + value); }
}
```

Here's the result of this code:

```
C:\>java App
The value = 1234567890123456789
```

You can also create literals in octal format by starting them with a leading zero and in hexadecimal format by starting with 0x or 0X. Here are some examples:

```
public class App { public static void main(String[] args) {
int value; value = 16;
System.out.println("16 decimal = " + value);
value = 020;
System.out.println("20 octal = " + value + " in decimal");
value = 0x10;
System.out.println("10 hexadecimal = " + value + " in decimal"); }
}
```

Here's what this program displays:

```
C:\>java App
16 decimal = 16
20 octal = 16 in decimal
10 hexadecimal = 16 in decimal
```

Creating Floating-Point Literals

The Novice Programmer appears and says, "I have a problem. I want to put a floating-point number 1.5 into a floating-point variable, but Java keeps saying 'Incompatible type for =. Explicit cast needed to convert double.' What's going on?" You say, "By default, the floating-point numbers used as literals are actually of type `double` and not of type `float`. You can change that by adding an *f* or *F* to the end of the literal to make it a `float`, or a *d* or *D* to make it a `double`." "Oh!" the NP replies.

Floating-point literals are of type `double` by default in the Java code; examples include 3.1415926535, 1.5, and 0.1111111. The standard notation for floating-point literals is to have a whole number followed by a fractional part. You can also indicate a power of 10 with *e* or *E*, like this:

```
1.345E10
```

This is the same as $1.345X10^{10}$ or –9.999E–23, which is the same as $-9.999X10^{-23}$. Here's an example in which we are trying to assign a floating-point literal to a variable of type `float`:

```
public class App { public static void main(String[] args) {
float value;
value = 1.5;
System.out.println("The value = " + value); }
}
```

Unfortunately, the default type for floating-point literals is `double`, so Java gives me the following error:

```
C:\>javac App.java
App.java:2: error: incompatible types: possible lossy conversion from double to float
value = 1.5;
        ^
 1 error
```

We can fix this by explicitly making my literal into a `float`, like this:

```
public class App {
        public static void main(String[] args) {
float value;
value = 1.5f;
```

```
    System.out.println("The value = " + value); }
    }
```

Now the code runs as you'd expect:

```
    C:\>java App
    The value = 1.5
```

Creating Boolean Literals

Boolean values can only be true or false in Java (not 0 or 1 or other numeric values as in other languages—this is the part of Java's strong data typing). This means that *only* two Boolean literals you can use are true and false. Here's an example using true as a Boolean literal:

```
    public class App {
            public static void main(String[] args) {
    boolean value;
    value = true;
    System.out.println("The value = " + value); }
    }
```

Here's the result of this program:

```
    C:\>java App
    The value = true
```

Creating Character Literals

"Hey," says the Novice Programmer, "how do I assign a letter to a variable in Java? I'm evaluating all the company's products and want to assign them letter grades". "You can use character literals, each of which represents a character," you reply. "By the way, does the Big Boss know about this?" "Not yet," says the NP.

The basic form of a Java character literal is just a value that corresponds to a character in the Unicode character set (for more on Unicode, see `http://www.unicode.org`). Character literals are actually numbers that act as indexes into the Unicode character set, not actual characters. For example, the Unicode code for the letter *C* is 67. Therefore, the following application prints out a C:

```
    public class App {
            public static void main(String[] args) {
    char char3;
    char3 = 67;
    System.out.println("The third letter in alphabet = " + char3); }
    }
```

Here's the result of the above code:

```
    C:\>java App
    The third letter in alphabet = C
```

However, you can also refer to the Unicode code for the letter *C* with a character literal, which you would enclose in single quotes, like this:

```
    public class App {
            public static void main(String[] args) {
    char char3; char3 = 'C';
    System.out.println("The third letter in alphabet = " + char3); }
    }
```

Here's is the result of this code, which, obviously, is the same as the result of the previous code:

```
    C:\>java App
    The third letter in alphabet = C
```

Besides enclosing characters in single quotes to make character literals, you can also enclose special *character escape sequences* in single quotes to make character literals that you couldn't make by typing a single character. Here are the escape sequences:

- `\'` (single quote)
- `\"` (double quote)
- `\\` (backslash)
- `\b` (backspace)

- \ddd (octal character)
- \f (form feed)
- \n (newline; called a *line feed* in DOS and Windows)
- \r (carriage return)
- \t (tab)
- \uxxxx (hexadecimal Unicode character)

For example, if you want to show a double quotation mark in the displayed text, you can use the \" escape sequence, like this:

```
public class App {
        public static void main(String[] args) {
   System.out.println("He said, \"Hello!\""); }
}
```

Here's the result of this code:

```
C:\>java App
He said, "Hello!"
```

Creating String Literals

The Novice Programmer is back—this time with some coffee. "OK", the NP says, "here's the problem: I want to use just one println statement to print out multiple lines—can I do that?" "Sure", you say, "As long as you use the \n character literal to stand for a newline". "How's that?" the NP asks.

Here's an example of what the NP wants to do. In this case, we'll print out some multiline text using the \n character literal to start a new line:

```
public class App {
        public static void main(String[] args) {
   System.out.println("Here is \nsome multiline\ntext"); }
}
```

Here's the output of this application:

```
C:\>java App
Here is
some multiline
text
```

As with most other programming languages, you can enclose text string literals in double quotes (unlike single character literals, which you enclose in single quotes). You can also use the character escape sequences introduced in the previous topic. Note that string literals in Java code are actually converted by the compiler into String objects, not inherent simple data types (which means that odd code such as "Hello".length() is perfectly legal and will return the length of the string "Hello").

Creating Binary Literals

The Novice Programmer is back and says, "How can we create a literal of integer type and save the value in the literal in binary format?" You are confused and say, "And why would you want to do that?" The NP says, "I have just learned the procedure to convert decimals to binary and thought that the computer eventually stores and processes everything in binary. Why not save some time of the computer by providing the value in binary directly. I want to help the poor guy."

One of the features that Java has implemented is to assign the values in binary format. If you are assigning a value in binary format, you need to use the letters 0b before keying in the actual value. The binary literals express the integral type in the binary number system.

The literals are useful in a number of programming tasks to be performed. They are frequently used to create numeric arrays. The relation between the data members of the array, which is represented using the binary literals, is more apparent than the relation represented using the members in the hexadecimal or octal notations. The reason behind the apparent relation representation is that the numbers represented using binary literals are

more detailed than the hexadecimal or octal literals. A good example of the relation between the array members is rotation of 2 bits among the successive array members, as shown in the following example:

```
public class App {
  public static void main(String[] args) {
int val = 0b00010000;
int  myPhases[] = {
0b00110011,
0b11001100,
0b00110011,
0b11001100 };
System.out.println("The value in val is: " + val);
System.out.println("And following are the values stored in the array myPhases:");
System.out.println(myPhases[0]);
System.out.println(myPhases[1]);
System.out.println(myPhases[2]);
System.out.println(myPhases[3]); }
}
```

The output of the program is as follows:

```
C:\>java App
The value in val is: 16
And following are the values stored in the array myPhases:
51
204
51
204
```

Using Underscores in Numeric Literals

The Novice Programmer says, "I want to store the credit card numbers in variables and make them more readable by separating the digits the way they are printed on the credit cards." The NP adds, "I would also like to perform mathematical operations on those values because I will be giving out the cards in sequential order." You respond, "Well, that is not possible unless you are using the latest version of Java, that is, Java 7 or 8." You add, "This is a new feature of Java 7 that allows the programmer to enter the integer values and separate them with underscores to make them more readable."

The use of underscores in numeric literals is a newly introduced feature of Java 7. This allows the programmer the freedom to use underscores in numeric literals, making it more readable and minimizing the typographical errors that are very likely otherwise. You can use any number of underscore characters (_) to break the groups of digits in the long numeric literals. The literals still remain very much ready for performing mathematical operations. This feature was added to Java just for the sake of adding more readability to the numeric values.

An example that demonstrates the usage of underscores in numeric literals is as follows:

```
public class App {
        public static void main(String[] args) {
long card = 1345_4489_4545_7252L;
System.out.println("Your Credit Card number is: "  + card);
  }
}
```

The output of the preceding code is as follows:

```
C:\>java App
Your Credit Card number is: 1345448945457252
```

The underscores will not be displayed during the execution, as they are only for the convenience of the programmer. For example, in the above program, we have created a variable of the type `long` and saved a hypothetical 16-digit credit card number that contains underscores to separate every four digits. The `L` at the end of the credit card number in the program signifies that the value is of the `long` data type.

It is important to note that you can put underscores in a numeric literal only in between the digits of the numeric literal. Or the underscore character should be accompanied by at least one digit on each side. You cannot place an underscore in the numeric literals in the following places:

- At the start or the end of the numeric literal
- Next to the decimal point in the floating-point literal
- In front of an `F` or `L` suffix in the floating-point or long literal
- Places where a string of digits is expected in the numeric literal

Declaring Integer Variables

"Now I'm into some real programming", the Novice Programmer says, "and I need to store some integer data. How can I do that?" "With an integer variable.", you say, "Pull up a chair and I'll go through it".

Java uses four types of integer, each with its own number of bytes put aside for it in memory—`byte` (1 byte), `short` (2 bytes), `int` (4 bytes), and `long` (8 bytes). For the range of possible values each type can handle, see the topic "What Data Types Are Available?" in the Immediate Solutions section of this chapter. Which one you use depends on the range of data you want to use as well as other considerations such as how much memory is available (in case you want to set up a lot of integers).

Here's an example that puts all the integer types to use, declares an integer of each type, assigns each type some data, and then displays that data:

```
public class App {
        public static void main(String[] args) {
byte byte1;
short short1;
int int1;
long long1;
byte1 = 1;
short1 = 100;
int1 = 10000;
long1 = 100000000;
System.out.println("byte1 = " + byte1);
System.out.println("short1 = " + short1);
System.out.println("int1 = " + int1);
System.out.println("long1 = " + long1); }
}
```

Here's the result of this application:

```
byte1 = 1
short1 = 100
int1 = 10000
long1 = 100000000
```

Declaring Floating-Point Variables

"Sorry," says the Novice Programmer, "but integers just don't cut it. I'm trying to design a currency converter, and I thought I could ignore the cents part of each value, but the Big Boss told me that every cent counts. Is there any other data type we can use?" "Sure," you say. "You can use the `float` and `double` types".

Java has two built-in types of floating-point variables, each with its own number of bytes set aside for it in memory: `float` (4 bytes) and `double` (8 bytes). For the range of possible values each type can handle, see the topic "What Data Types Are Available?" in this chapter. The one you use depends on the range of data you want to use as well as other considerations such as how much memory is available (in case you want to set up a lot of floating-point values).

Here's an example that declares and uses both a `float` and a `double` (note that we explicitly make each literal value either a `float` or a `double` literal so that there'll be no problem with type conversions):

```
public class App {
        public static void main(String[] args) {
  float float1;
```

```
        double double1;
        float1 = 1.11111111111F;
        double1 = 1.1111111111111E+9D;
        System.out.println("float1 = " + float1);
        System.out.println("double1 = " + double1); }
    }
```

Here's the output of the code (note that we have exceeded the precision allowed for a `float`, so its value is rounded):

```
C:\>java App
float1 = 1.1111112
double1 = 1.1111111111111E9
```

Declaring Character Variables

You can declare character variables with the keyword `char`. For the possible values you can store in a `char` variable, see the topic "Creating Character Literals" in this chapter.

Here's an example that declares two `char` variables: `char1` and `char2`. This example demonstrates that you can assign either a Unicode code or a character literal to a `char` (in fact, the compiler translates character literals into Unicode codes):

```
public class App {
        public static void main(String[] args) {
char char1, char2;
char1 = 65;
char2 = 'B';
System.out.println("char1 = " + char1);
System.out.println("char2 = " + char2); }
}
```

Here's the result of this code:

```
C:\>java App
char1 = A
char2 = B
```

Here's a sneak peak at a future topic, in which we add some text to the end of `char1`, converting it to a string, and increment the value in `char2`, changing it from 'B' to 'C':

```
public class App {
        public static void main(String[] args) {
char char1, char2;
char1 = 65;
char2 = 'B';
System.out.println("char1 = " + char1);
System.out.println("char2 = " + char2);
System.out.println("char1 + 1 = " + char1 + 1);
System.out.println("char2 = " + ++char2); }
}
```

Here's the output of the new version of this program:

```
C:\>java App
char1 = A
char2 = B
char1 + 1 = A1
char2 = C
```

Declaring Boolean Variables

You declare Boolean variables with the `boolean` keyword. Boolean variables can take only two values in Java—true and false (not numerical values such as 0 and 1 as in other programming languages). Here's an example in which we declare and use two Boolean variables:

```
public class App {
        public static void main(String[] args) {
boolean boolean1, boolean2;
boolean1 = true;
boolean2 = false;
```

```
System.out.println("boolean1 = " + boolean1);
System.out.println("boolean2 = " + boolean2);
}
}
```

Here's the result of this code:

```
C:\>java App
boolean1 = true
boolean2 = false
```

Boolean values are usually used in tests to determine program flow. We are going to jump the gun here and give you a sneak peek into the next chapter. Here, we are using these two Boolean variables with the Java `if` statement. We test the value in `boolean1` with the `if` statement, which makes the code display the message "boolean1 is true" (if it's true) and "boolean1 is false" (otherwise):

```
public class App {
        public static void main(String[] args) {
  boolean boolean1, boolean2;
  boolean1 = true;
  boolean2 = false;
  System.out.println("boolean1 = " + boolean1);
  System.out.println("boolean2 = " + boolean2);
  if(boolean1) {System.out.println("boolean1 is true"); }
  else { System.out.println("boolean1 is false"); }
        }
}
```

Here's the new result from this code:

```
C:\>java App
boolean1 = true
boolean2 = false
boolean1 is true
```

Initializing Variables Dynamically

"OK", the Novice Programmer says, "I have got it straight now. First, I declare a variable and then I assign a value to it". "Actually", you say, "You can do both in one step". The NP replies, "Tell me how!"

So far, we have been declaring variables and then assigning values to them, like this:

```
public class App {
        public static void main(String[] args) {
int int1;
int1 = 1;
System.out.println("int1 = " + int1); }
}
```

However, we can combine these two steps into one by initializing a variable when we declare it, like this:

```
public class App {
        public static void main(String[] args) {
int int1 = 1;
System.out.println("int1 = " + int1); }
}
```

Here's how to declare and initialize multiple variables:

```
public class App {
        public static void main(String[] args) {
int int1 = 1, int2 = 2, int3 = 3;
System.out.println("int1 = " + int1 + ", int2 = " + int2 + ", int3 = " + int3); }
}
```

Here's the result of this program:

```
C:\>java App
int1 = 1, int2 = 2, int3 = 3
```

Up to this point, we have just assigned constant values to variables, but you can assign any expression (an *expression* is any combination of Java terms that yields a value) to a variable when that variable is declared, as long as the expression is valid at that time. For example, here we are assigning a value of 2 to `int1`, a value of 3 to `int2`, and the value of `int1` times `int2` to `int3` using the Java multiplication operator (*):

```
public class App {
        public static void main(String[] args) {
int int1 = 2, int2 = 3;
int int3 = int1 * int2;
System.out.println("int1 = " + int1 + ", int2 = " + int2 +
", int3 = " + int3); }
}
```

Here's what this code gives you when you run it:

```
C:\>java App
int1 = 2, int2 = 3, int3 = 6
```

Note that the Java compiler has no idea what `int1` times `int2` will be when it creates the *bytecodes* for this application. This means the actual value with which `int3` is initialized will be determined at runtime, which is why this process is called *dynamic initialization*.

As in C++, in Java, you can also intersperse your variable declarations throughout your code, as we are doing here:

```
public class App {
        public static void main(String[] args) {
int int1 = 2, int2 = 3;
System.out.println("int1 = " + int1 + ", int2 = " + int2);
int int3 = int1 * int2;
System.out.println("int3 = " + int3); }
}
```

Here's the result of this code:

```
C:\>java App
int1 = 2, int2 = 3
int3 = 6
```

Conversion between Data Types

"Uh oh", says the Novice Programmer. "I'm stuck. I have an `int` variable that I want to assign to a byte variable, but Java keeps giving me an 'Incompatible type for =.' error. What's wrong?" "That's a type conversion problem", you explain, "and you have to use an explicit type cast". "Hmm", says the NP, "how does that work?"

Java is a strongly typed language; and as a result, you are often faced with the situation of assigning a variable of one type to a variable of another type. The assigning of a variable of one type to a variable of another type can be done by either relying on automatic type conversion or making an explicit type cast. We'll take a look at both here.

Automatic Conversions

When you are assigning one type of data to a variable of another type, Java will convert the data to the new variable type automatically if both the following conditions are true:

- The data type and the variable types are compatible.
- The target type has a larger range than the source type.

For example, you can assign a `byte` value to an `int` variable because `byte` and `int` are compatible types, and `int` variables have a larger range than `byte` values. Therefore, no data will be lost in the type conversion. Here's an example:

```
public class App {
        public static void main(String[] args) {
byte byte1 = 1;
int int1;
int1 = byte1;
System.out.println("int1 = " + int1); }
}
```

The Java compiler has no problem with this code, and it makes the type conversion automatically. Here's the result of this program:

```
 C:\>java App
int1 = 1
```

Converting a data type to another with a larger range is called *widening conversion*. In widening conversions, the numeric types, such as the integer and floating-point types, are compatible with each other. On the other hand, `char` and `boolean` types are not compatible with each other or with the numeric types.

Casting to New Data Types

Performing a conversion from a data type of larger range to one that has a smaller range is called *narrowing conversion*. The Java compiler does not perform automatic narrowing conversions because there are chances of losing accuracy. To perform a narrowing conversion, it must be done with an explicit cast, which looks like this:

```
(target-data-type) value
```

For example, in this code, we are converting an `integer` type into a `byte` type:

```
public class App {
        public static void main(String[] args) {
byte byte1; int int1 = 1;
byte1 = (byte) int1;
System.out.println("byte1 = " + byte1); }
}
```

Trying to perform a narrowing conversion would throw an exception, but performing the same with the explicit type cast would not. This is due to the fact that Java assumes that you are aware of the possibility of losing data by converting a potentially larger data with a larger data type into a smaller type. Simply put, the programmer is responsible for any loss of data while performing any narrowing conversions. For example, when you put a floating-point number into a `long`, the fractional part of the number will be truncated, and you may lose more data if the floating-point value is outside the range that a `long` can hold.

Here's the output of this code:

```
C:\>java App
byte1 = 1
```

One thing to note is that the Java compiler also automatically promotes types as needed when it evaluates expressions. For example, consider the following code, in which everything looks like it involves only bytes:

```
public class App {
  public static void main(String[] args) {
byte byte1 = 100; byte byte2 = 100; byte byte3;
byte3 = byte1 * byte2 / 100;
System.out.println("byte3 = " + byte3); }
}
```

Here Java gives the following error:

```
C:\>javac App.java
App.java:4: error: incompatible types: possible lossy conversion from int to byte
byte3 = byte1 * byte2 / 100;
                      ^
1 error
```

However, because Java knows that multiplying bytes can result in integer-sized values, it automatically promotes the result of the `byte1 * byte2` operation to an integer, which means you actually have to use an explicit cast here to get back to the `byte` type:

```
public class App {
        public static void main(String[] args) {
byte byte1 = 100;
byte byte2 = 100;
byte byte3;
byte3 = (byte) (byte1 * byte2 / 100);
System.out.println("byte3 = " + byte3); }
}
```

This code compiles and runs as you'd expect—but it wouldn't without the (`byte`) cast:

```
C:\>java App
byte3 = 100
```

TIP

In general, the Java compiler promotes byte and short types to int types in expressions. If one operand is a long, the entire expression is made a long. Similarly, if one operand is a float, the whole expression is made a float; if one operand is a double, the whole expression is made a double.

Declaring One-Dimensional Arrays

The Big Boss appears and says, "It's time to start cracking down on customers who are overdue paying their bills to me". "OK", you say, "Can I see the accounts?" "I never actually kept any accounts", the BB says. "Oh", you reply. "I guess I'll have to set up an array to store the accounts in first".

As explained earlier in this chapter, arrays provide an easy way of handling a set of data by index, which is great for computers because then you can manipulate the index in your code. Java supports one-dimensional and multi-dimensional arrays, and we'll take a look at both of them here. Getting an array ready for use is a two-step process. First, you must declare the array. Here's how you declare a one-dimensional array in general:

```
type name[];
```

For example, here's how to declare an array of double values, which we'll name `accounts[]`:

```
public class App {
        public static void main(String[] args) {
double accounts[];
. . . }
  }
```

TIP

In fact, there's another way of doing this that follows the pointer-declaration syntax in C++. You can also declare arrays with the brackets ([]) after the type, not the name of the variable, like this: double[ ] accounts.

Unlike declaring simple variables, declaring an array does not set aside memory for the array because Java isn't sure how big you want it to be yet. This means there's another step to the process—actually creating the array. See the next topic for the details.

Creating One-Dimensional Arrays

After you've declared a one-dimensional array, the next step is to actually create that array by allocating memory for it. As you'll see in the forthcoming chapter, the Java memory allocation operator is the `new` operator. Therefore, we can create and use the `new` array like this:

```
public class App {
        public static void main(String[] args) {
double accounts[];
accounts = new double[100];
accounts[2] = 1335.67;
System.out.println("Account 3 is overdue by $" + accounts[2]); }
}
```

Here, we have created an array of exactly 100 `double` values, all of which Java initializes to 0. The lower bound of all Java arrays is 0, so the first element in the array is `accounts[0]` and the last element is `accounts[99]`. If the array index is outside the range 0 to 99, Java will create a fatal error, and the program will halt. Here's the result of this program:

```
C:\>java App
Account 3 is overdue by $1335.67
```

In fact, you can combine the declaration and creation processes into one step for arrays, like this:

```
public class App {
        public static void main(String[] args) {
double accounts[] = new double[100];
accounts[2] = 1335.67;
System.out.println("Account 3 is overdue by $" + accounts[2]); }
}
```

Initializing One-Dimensional Arrays

The Novice Programmer is back with a question. "I know I can initialize simple variables when I declare them", the Novice Programmer says, "but what about initializing arrays when I declare them?" "No problem", you say.

To initialize the data in one-dimensional arrays, you just place the values between curly braces, one value after the other, separated by commas, beginning with the first value in the array. Here's an example that initializes the first four elements of the `accounts[]` array with data:

```
public class App {
        public static void main(String[] args) {
double accounts[] = {238.45, 999.33, 0, 1335.67};
System.out.println("Account 4 is overdue by $" + accounts[3]); }
}
```

Declaring Multi-Dimensional Arrays

"Hmm," says the Novice Programmer thoughtfully, "I think I need more than a one-dimensional array. I'm supposed to be keeping track of products as indexed by product number, and the array is supposed to store the number of items in inventory, the cost of each item, the number sold, the number …." "Hold it," you say. "Use a multi-dimensional array."

You can declare multi-dimensional arrays in much the same way you declare one-dimensional arrays; just include a pair of square brackets for every dimension in the array:

```
type name[][][]...;
```

We looked at declaring multi-dimensional arrays previously in this chapter; for example, here's how to declare a two-dimensional array with two rows and 100 columns:

```
public class App {
        public static void main(String[] args) {
double accounts[][] = new double[2][100];
. . . }
}
```

TIP

In fact, there's another way of doing this that follows the pointer-declaration syntax in C++. You can also declare arrays with the brackets ([ ]) after the type, not the name of the variable, like this: double[ ][ ] accounts.

That's how it works with two-dimensional arrays—the left index specifies the row in the array and the right index specifies the column. Of course, you don't have to limit yourself to two dimensions.

Here's how you can declare a four-dimensional array:

```
public class App {
        public static void main(String[] args) {
double accounts[][][][] = new double[2][3][4][5];
. . . }
}
```

As you can see, it's as easy to declare multi-dimensional arrays as it is to declare one-dimensional arrays. Now what about actually creating the declared array? See the next topic for the details.

Creating Multi-Dimensional Arrays

The Novice Programmer asks, "Now that I have declared a new multi-dimensional array, how do I *create* it?" "Coming right up," you say.

You create a new multi-dimensional array by allocating memory for it with the new operator, giving the array the dimensions you want. Here's how this looks in an example:

```
public class App {
        public static void main(String[] args) {
double accounts[][];
accounts = new double[2][100];
accounts[0][2] = 43.95;
accounts[1][2] = 2385489382.06;
System.out.println("Savings account 3 has $" + accounts[0][2]);
```

```
    System.out.println("Checking account 3 has $" + accounts[1][2]); }
    }
```

Here's the result of this code:

```
C:\>java App
Savings account 3 has $43.95
Checking account 3 has $2.38548938206E9
```

You can also condense the declaration and memory allocations into one step, like this:

```
public class App {
        public static void main(String[] args) {
double accounts[][] = new double[2][100];
accounts[0][2] = 43.95;
accounts[1][2] = 2385489382.06;
System.out.println("Savings account 3 has $" + accounts[0][2]);
System.out.println("Checking account 3 has $" + accounts[1][2]); }
}
```

Here's an example that creates and uses a four-dimensional array:

```
public class App {
        public static void main(String[] args) {
double accounts[][][][] = new double[2][3][4][5];
accounts[0][1][2][3] = 43.95;
System.out.println("Account [0][1][2][3] has $" +
accounts[0][1][2][3]); }
}
```

Here's the result of this program:

```
C:\>java App
Account [0][1][2][3] has $43.95
```

Multi-dimensional arrays are actually arrays of arrays, which means that if you have a two-dimensional array (`array[][]`), you can actually treat it as an array of one-dimensional arrays, which you can access as `array[0]`, `array[1]`, `array[2]`, and so on. Here's a slightly advanced example of doing just that using a `for` loop (which you'll see more of in the next chapter) and using the `length` property (which you'll see in a few topics) to find the length of an array:

```
public class App {
        public static void main(String[] args) {
   double array[][] = {{1, 2, 3}, {3, 2, 1}, {1, 2, 3}};
   int sum = 0, total = 0;
   for(int outer_index = 0; outer_index < array.length; outer_index++){
      for(int inner_index = 0; inner_index < array[outer_index].length; inner_index++) {
                sum += array[outer_index][inner_index];
                total++; } }
   System.out.println("Average array value = " + (sum / total)); }
}
```

Here's the result of this code:

```
C:\>java App
Average array value = 2
```

So far, all the arrays we have used have had the same number of elements for each dimension, but you don't need to set things up that way. To learn more, you can take a look at the topic after the next one. First let's look at initializing multi-dimensional arrays.

Initializing Multi-Dimensional Arrays

You can initialize multi-dimensional arrays with data when declaring them in much the same way you initialize one-dimensional arrays—just include a set of curly braces for each dimension and place the values with which you want to initialize the array in those curly braces. For example, here's how to initialize a two-dimensional array:

```
public class App {
        public static void main(String[] args) {
   double accounts[][] = { {10.11, 19.56, 4343.91, 43.95},
                { 11.23, 54.23, 543.62, 2385489382.06 } };
   System.out.println("Savings account 3 has $" + accounts[0][2]);
   System.out.println("Checking account 3 has $" +
```

```
      accounts[1][2]); }
   }
```

Here's what running this code yields:

```
   C:\>java App
   Savings account 3 has $4343.91
   Checking account 3 has $543.62
```

Creating Irregular Multi-Dimensional Arrays

"OK", says the Novice Programmer proudly. "Now I'm an array expert". "Uh huh", you say. "Can you give each row in an array a different number of elements?" The NP says, "Excuse me?"

As with many other programming languages, multi-dimensional arrays are actually arrays of arrays in Java. This means you can construct arrays as you like, as in the below example, in which each row of a two-dimensional array has a different number of elements:

```
   public class App {
            public static void main(String[] args) {
   double array[][] = new double[5][];
   array[0] = new double[500]; array[1] = new double[400];
   array[2] = new double[300]; array[3] = new double[200];
   array[4] = new double[100]; array[3][3] = 1335.67;
   System.out.println("Account [0][400] has $" + array[0][400]);
   System.out.println("Account [1][300] has $" + array[1][300]);
   System.out.println("Account [2][200] has $" + array[2][200]);
   System.out.println("Account [3][100] has $" + array[3][100]);
   System.out.println("Account [4][50] has $" + array[4][50]);
   System.out.println("Account [3][3] has $" + array[3][3]); }
   }
```

What's happening here is that we are treating each row of a two-dimensional array as a one-dimensional array by itself and creating each of those one-dimensional arrays separately.

Each dimensional array defined in this program fetches the element according to the value of index taken individually. Note here that the value of the array variable that is not defined is taking the default value; as in this case, the default value of double is 0.0.

Here's the result of this code:

```
   C:\>java App
   Account [0][400] has $0.0
   Account [1][300] has $0.0
   Account [2][200] has $0.0
   Account [3][100] has $0.0
   Account [4][50] has $0.0
   Account [3][3] has $1335.67
```

Now, let's fetch the value that exceeds the array index as shown in the following program:

```
   public class App {
            public static void main(String[] args) {
   double array[][] = new double[5][];
   array[0] = new double[500];
   array[1] = new double[400];
   array[2] = new double[300];
   array[3] = new double[200];
   array[4] = new double[100];
   array[3][3] = 1335.67;
   System.out.println("Account [0][500] has $" + array[0][500]);
   System.out.println("Account [1][500] has $" + array[1][500]);
   System.out.println("Account [2][400] has $" + array[2][400]);
   System.out.println("Account [3][300] has $" + array[3][300]);
   System.out.println("Account [4][200] has $" + array[4][200]);
   System.out.println("Account [3][3] has $" + array[3][3]); }
   }
```

The preceding program gets compiled but running the code yields, is the exceptional handling error as follows:

```
C:\>java App
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 500
        at App.main(App.java:10)
```

Getting an the Length of an Array

It's often useful to know the length of an array, especially if you are iterating over all elements in the array in your code. To find the number of elements in an array named `array1`, you can use the term `array1.length`. Here's an example from the next chapter that uses a `for` loop to find the average student grade from a set of six grades (here, the term `grades.length` returns a value of 6):

```
public class App {
         public static void main(String[] args) {
double grades[] = {88, 99, 73, 56, 87, 64};
double sum, average;
sum = 0;
for (int loop_index = 0; loop_index < grades.length; loop_index++) {
         sum = sum + grades[loop_index]; }
  average = sum / grades.length;
  System.out.println("Average grade = " + average); }
 }
```

Here's the result of this code:

```
C:\>java App
Average grade = 77.83333333333333
```

Understanding General Form of Static Import

To access static members, it is required to qualify references with the class they belong to. For example,

```
double calc = Math.cos(Math.PI * theta);
```

To understand this better, you sometimes add static members to an interface and inherit the same from that interface. This is not advised though, because a class of the static members of another class you are using is just a plain implementation detail. Whenever a class implements an interface, it becomes the part of the class's public API, and implementation detail should not interfere with other public APIs.

The static import allows unqualified access to static members. You do not need to *inherit* from the type containing static members. Alternatively, the program may import the members individually:

```
import static java.lang.Math.PI;
```

or collectively:

```
import static java.lang.Math.*;
```

The static members may be used without criterion, once imported:

```
double calc = cos(PI * theta);
```

The static import declaration mentioned previously and the normal import declaration are comparable.

You can use the static import instead of declaring local copies of constants, or utilizing inheritance (the `Constant` Interface Antipattern). This means that the static import is used when we require frequent access to static members from a couple of classes. However, excess use of this feature can make your program unreadable and difficult to maintain, creating problems with the namespace of all the static members you import.

Importing Static Members

In a program, you can import the static members of a class by simply importing the named package of that class. For example, in the `Circumference` class, you can use the constant `PI` defined in the `Math` class by using either its fully qualified name `Math.PI` or simply `PI` if you import the `PI` member of the `Math` package:

```
double circumference_circle() { return 2 * Math.PI*radius; }
```

This shows the need for the static member of the `Circumference` class with the name `PI` and would provide a much more accurate definition of π. The `Math` prefix to the name `PI` doesn't really add clarity to the code, and it would be better without it.

We can avoid the prefixing of `PI` to the `Math` class name by importing the `PI` member name from the `Math` class:

```
import static java.lang.Math.PI;
class Circumference {
   . . .
   double circumference_circle() { return 2*PI*radius; }
}
```

All the static members can also be imported by using the `*` notation. This import applies only to the classes defined in a named package for the static members of a class. If you require importing the name of static member of a class that you define, the definition of the class should be included in a named package. Names of static members with no names in the default package cannot be imported. The class name in the static import statement must always be qualified with its package name.

The **String** Class

"I have been looking through the list of simple data types in Java", the Novice Programmer says, "And I can't find text strings there. Shouldn't they be there?" "Some people say so", you reply. "Strings are handled as objects in Java. One advantage of this is that a `String` object has a great variety of methods you can use with it".

In many languages, text strings are fundamental data types inherent to the specific language; but in Java, strings are handled with the `String` and `StringBuffer` classes. Let's take a look at the `String` class first.

`String` objects hold text strings that you can't change; if you want to change the actual text in the string, you should use the `StringBuffer` class instead. Here's an example in which we create a string and print it out (notice how much this code makes the `String` class look like any other simple data type):

```
public class App {
        public static void main(String[] args) {
String s1 = "Hello from Java!";
System.out.println(s1); }
}
```

Here's the result of this code:

```
C:\>java App
Hello from Java!
```

The `String` class is enormously powerful, with methods that enable you to convert the string into a character array, convert numbers into strings, search strings, create substrings, change the case of the string, get a string length, compare strings, and much more. The `String` class is obviously a class, not an intrinsic data type, which means you create objects of that class with constructors, which you'll learn all about in Chapter 4. A *constructor* is just like a normal method of a class, except you use it to create an object of that class. You'll get a sneak peek at the `String` class constructors here. The `String` class also has a data member you use when comparing strings (which we'll take a look at in the next chapter). This data member is shown in Table 2.2. The `String` class constructors, which you can use to create `String` objects (see the topic "Creating Strings", coming up in this chapter), appear in Table 2.3, and the methods of the `String` class appear in Table 2.4.

We'll put the material you see in the tables below to use in the next few topics, in which we'll create and use `String` objects.

Table 2.2: String class field summary

Field	Does this
`static Comparator<String> CASE_INSENSITIVE_ORDER`	It yields a comparator (which you'll see more about later) that orders `String` objects, as in `compareToIgnoreCase`

Table 2.3: String class constructor summary

Constructor	Does this
`String()`	It initializes a new `String` object so that it holds an empty character sequence
`String(byte[] bytes)`	It constructs a new `String` object by converting the array of bytes by using the platform's default character encoding
`String(byte[] bytes, Charset charset)`	It constructs a new `String` by decoding the specified array of bytes using the specified charset
`String(byte[] ascii, int hibyte)`	It is deprecated because it did not properly convert bytes into characters
`String(byte[] bytes, int offset, int length)`	It constructs a new `String` object by converting the subarray of bytes using the default character encoding
`String(byte[] bytes, int offset, int length, Charset charset)`	It constructs a new `String` by decoding the specified subarray of bytes using the specified charset
`String(byte[] ascii, int hibyte, int offset, int count)`	It is deprecated because it did not properly convert bytes into characters
`String(byte[] bytes, int offset, int length, String charsetName)`	It constructs a new `String` object by converting the subarray of bytes using the specified charset
`String(byte[] bytes, String charsetName)`	It constructs a new `String` object by converting the array of bytes using the specified charset
`String(char[] value)`	It allocates a new `String` object so that it represents the sequence of characters contained in the character array argument
`String(char[] value, int offset, int count)`	It allocates a new `String` object that contains characters from a subarray of the character array argument
`String(int[] codePoints, int offset, int count)`	It allocates a new `String` object that contains characters from a subarray of the Unicode code point array argument
`String(String original)`	It initializes a new `String` object so that it represents the same sequence of characters as the argument string
`String(StringBuffer buffer)`	It allocates a new `String` object that contains the sequence of characters contained in the string buffer argument
`String(StringBuilder builder)`	It allocates a new `String` object that contains the sequence of characters contained in the string builder argument

Table 2.4: String class methods

Method	Does this
`char charAt(int index)`	It yields the character at the given index
`int codePointAt (int index)`	It yields the character (Unicode code point) at the specified index
`int codePointBefore(int index)`	It yields the character (Unicode code point) before the specified index
`int codePointCount(int beginIndex, int endIndex)`	It yields the number of Unicode code point in the specified range of this string
`int compareTo (String anotherString)`	It compares two strings lexicographically
`int compareToIgnoreCase (String str)`	It compares two strings lexicographically, ignoring case
`String concat(String str)`	It concatenates the given string to the end of this string
`boolean contains(CharSequence s)`	It returns True if this string contains the specified sequence of char values

Table 2.4: String class methods

Method	Does this
`boolean contentEquals(CharSequence cs)`	It compares this string to the specified `CharSequence`
`boolean contentEquals(StringBuffer sb)`	It compares this string to the specified `StringBuffer`
`static String copyValueOf (char[] data)`	It yields a `String` object that's equivalent to the given character array
`static String copyValueOf (char[] data, int offset, int count)`	It yields a `String` object that's equivalent to the given character array by using offsets
`boolean endsWith (String suffix)`	It returns True if the string ends with the given suffix
`boolean equals (Object anObject)`	It compares this string to an object
`boolean equalsIgnoreCase (String anotherString)`	It compares this `String` object to another `String` object, ignoring case
`static String format(Locale l, String format, Object...args)`	It returns a formatted string using the given locale, format string, and arguments
`static String format(String format, Object...args)`	It returns a formatted string using the given format string and arguments
`byte[] getBytes()`	It converts this `String` object into bytes with the default charset, storing the result in a new byte array
`byte[] getBytes(Charset charset)`	It encodes this String into a byte sequence using the given charset, storing the result in a new byte array
`void getBytes (int srcBegin,int srcEnd, byte[] dst, int dstBegin)`	It is deprecated because it did not properly convert characters into bytes
`byte[] getBytes (String charsetName)`	It converts this `String` object into bytes with the given charset, storing the result in a new byte array
`void getChars (int srcBegin, int srcEnd, char[] dst, int dstBegin)`	It copies characters from this string into the destination array
`int hashCode()`	It yields a hash code for this string
`int indexOf(int ch)`	It yields the index within this string of the first occurrence of the given character
`int indexOf(int ch, int fromIndex)`	It yields the index within this string of the first occurrence of the given character, starting the search at the given index
`int indexOf(String str)`	It yields the index within this string of the first occurrence of the given substring
`int indexOf(String str, int fromIndex)`	It yields the index within this string of the first occurrence of the given substring, starting at the given index
`String intern()`	It yields a representation for the `String` object
`boolean isEmpty()`	It yields True if length of string is 0
`int lastIndexOf(int ch)`	It yields the index within this string of the last occurrence of the given character
`int lastIndexOf(int ch, int fromIndex)`	It yields the index within this string of the last occurrence of the given character, searching backward from the given index
`int lastIndexOf(String str)`	It yields the index within this string of the rightmost occurrence of the given substring

Table 2.4: String class methods

Method	Does this
`int lastIndexOf (String str,int fromIndex)`	It yields the index within this string of the last occurrence of the given substring, searching backward from the given index
`int length()`	It yields the length of this string
`boolean matches(String regex)`	It returns True if this string matches the given regular expression
`int offsetByCodePoints(int index, int codePointOffset)`	It returns the index within this string which is offset from the given `index` by `codePointOffset` code points
`boolean regionMatches (boolean ignoreCase, int toffset, String other, int ooffset, int len)`	It tests whether two string regions are equal, allowing you to ignore case
`boolean regionMatches(int toffset, String other, int ooffset, int len)`	It tests whether two string regions are equal
`String replace(char oldChar, char newChar)`	It yields a new string by replacing all occurrences of `oldChar` in this string with `newChar`
`String replace(CharSequence target, CharSequence replacement)`	It yields a new string by replacing each substring of this string that matches the literal target sequence with the given literal replacement sequence
`String replaceAll(String regex, String replacement)`	It yields a new string by replacing each substring of this string that matches the given regular expression with the given replacement
`String replaceFirst(String regex, String replacement)`	It yields a new string by replacing the first substring of this string that matches the given regular expression with the given replacement
`String[] split(String regex)`	It splits this string around matches of the given regular expression
`String[] split(String regex, int limit)`	It splits this string around matches of the given regular expression and the limit given as parameter
`boolean startsWith(String prefix)`	It tests whether this string starts with the given prefix
`boolean startsWith(String prefix, int toffset)`	It tests whether this string starts with the given prefix, beginning at the given index
`CharSequence subSequence( int beginIndex, int endIndex)`	It returns a new character sequence, which is a subsequence of this sequence
`String substring (int beginIndex)`	It yields a new string, which is a substring of this string
`String substring(int beginIndex, int endIndex)`	It yields a new string, which is a substring of this string, allowing you to specify the end index
`char[] toCharArray()`	It converts this string to a new character array
`String toLowerCase()`	It converts all the characters in this `String` object to lowercase by using the rules of the default locale, which is returned by `Locale.getDefault() method`
`String toLowerCase (Locale locale)`	It converts all the characters in this `String` object to lowercase using the rules of the given `Locale`
`String toString()`	It returns the string object
`String toUpperCase()`	It converts all the characters in this String object to uppercase using the rules of the default locale, which is returned by `Locale.getDefault() method`
`String toUpperCase (Locale locale)`	It converts all the characters in this `String` object to uppercase using the rules of the given locale
`String trim()`	It removes white space from both ends of this string

Table 2.4: String class methods

Method	Does this
`static String valueOf (boolean b)`	It yields the string representation of the `boolean` argument
`static String valueOf (char c)`	It yields the string representation of the `char` argument
`static String valueOf (char[] data)`	It yields the string representation of the `char` array argument
`static String valueOf (char[ ] data, int offset, int count)`	It yields the string representation of a specific subarray of the `char` array argument
`static String valueOf(double d)`	It yields the string representation of a `double` type argument
`static String valueOf(float f)`	It yields the string representation of a `float` type argument
`static String valueOf (int i)`	It yields the string representation of an `int` type argument
`static String valueOf(long l)`	It yields the string representation of a `long` type argument
`static String valueOf(Object obj)`	It yields the string representation of an `Object` type argument

Creating Strings

"So Java includes a `String` class to handle text strings", the Novice Programmer says. "That's great, because I'm writing this novel, see, and…". "Hold it", you say. "I don't want to hear about it".

Let's take a look at some of the many ways of creating `String` objects. Here's a way you've already seen:

```
public class App { public static void main(String[] args) {
String s1 = "Hello from Java!";
. . . }
}
```

In fact, when you use a string literal such as "Hello from Java!" in your code, Java treats it as a `String` object So what's really happening here is that one `String` object is assigned to another.

Of course, you can also declare a string first and then assign a value to it:

```
public class App { public static void main(String[] args) {
String s1 = "Hello from Java!";
String s2;
s2 = "Hello from Java!";
. . . }
}
```

Here's a case in which we use one of the `String` class constructors. In this case, we are just creating an empty string and then assigning data to it:

```
public class App {
        public static void main(String[] args) {
String s1 = "Hello from Java!";
String s2;
s2 = "Hello from Java!";
String s3 = new String();
s3 = "Hello from Java!";
. . .}
}
```

You can also pass a text string to the `String` class constructor directly to create a new string, like this:

```
public class App {
        public static void main(String[] args) {
String s1 = "Hello from Java!";
String s2;
s2 = "Hello from Java!";
```

```
String s3 = new String();
s3 = "Hello from Java!";
String s4 = new String("Hello from Java!");
. . . }
}
```

Other `String` class constructors are available that can take character arrays or subsets of character arrays (the `String` class knows which constructor you are using by the number and type of arguments you pass to it). You can even use the `String` class `valueOf` method to get a string representation of numeric values:

```
public class App {
        public static void main(String[] args) {
String s1 = "Hello from Java!";
String s2;
s2 = "Hello from Java!";
String s3 = new String();
s3 = "Hello from Java!";
String s4 = new String("Hello from Java!");
char c1[] = {'H', 'i', ' ', 't', 'h', 'e', 'r', 'e'};
String s5 = new String(c1);
String s6 = new String(c1, 0, 2);
String s7 = String.valueOf(c1);
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
System.out.println(s4);
System.out.println(s5);
System.out.println(s6);
System.out.println(s7);   }
}
```

TIP

To convert a string into a number, you can use the numeric wrapper classes, such as Integer, Long, Float, and so on, using methods such as Integer.parseInt() and Long.parseLong().

At the end of this code, we print out all the strings we have created. Here's what appears when the program is run:

```
C:\>java App
Hello from Java!
Hello from Java!
Hello from Java!
Hello from Java!
Hi there
Hi
Hi there
```

Getting String Length

The Novice Programmer is breathless. "I have written half my novel", the NP says, "and I need to find out how long it is so far. How can I do that?" "Use the `String` class `length()` method", you say.

Here's an example that shows how to use the `String` class's `length()` method (note that it also shows how Java treats string literals as `String` objects by using `length()` method on a string literal):

```
public class App {
        public static void main(String[] args) {
String s1 = "Hello from Java!";
System.out.println("\"" + s1 + "\"" + " is " + s1.length() + " characters long");
System.out.println("\"" + "Hello" + "\"" + " is " + "Hello".length() + " characters
  long"); }
}
```

Here's the output of this program:

```
C:\>java App
"Hello from Java!" is 16 characters long
"Hello" is 5 characters long
```

Concatenating Strings

Concatenating strings means joining them together, and we have already used the + operator in this book to do just that. However, there's another way to concatenate strings—you can use the `String` class `concat()` method to join two strings and create a new one.

How does that look in code? Here's an example where we use both the + operator and the `concat()` method to create the same string:

```
public class App {
        public static void main(String[] args) {
String s1 = "Hello";
String s2 = s1 + " from";
String s3 = s2 + " Java!";
String s4 = s1.concat(" from");
String s5 = s4.concat(" Java!");
System.out.println(s3);
System.out.println(s5);   }
}
```

Here's the result of the preceding code:

```
C:\>java App
Hello from Java!
Hello from Java!
```

As you have already seen when printing out numbers, when you concatenate a numeric value with a string, the numeric value is concatenated as a string.

TIP

Note that concatenating numbers does indeed treat them as strings, so be careful—for example, System.out.println ("3 + 3 = " + 3 + 3) displays 3 + 3 = 33, not 3 + 3 = 6.

Getting Characters and Substrings

The `String` class provides a number of methods that let you dissect strings into their component characters and substrings. For example, you can use the `charAt()` method to get the character at a specific position:

```
  public class App {
        public static void main(String[] args) {
String s1 = "Hello from Java!";
char c1 = s1.charAt(0);
System.out.println("The first character of \"" + s1 + "\" is " + c1);
. . . }
}
```

You can use the `toCharArray()` method to convert a `String` object into a `char` array, and you can use the `getChars()` method to get the number of characters:

```
public class App {
        public static void main(String[] args) {
String s1 = "Hello from Java!";
char c1 = s1.charAt(0);
System.out.println("The first character of \"" + s1 + "\" is " + c1);
char chars1[] = s1.toCharArray();
System.out.println("The second character of \"" + s1 + "\" is " + chars1[1]);
char chars2[] = new char[5];
s1.getChars(0, 5, chars2, 0);
```

```
System.out.println("The first five characters of \"" + s1 + "\" are " + new
  String(chars2));
. . .
```

You can also use the `substring()` method to create a new string that's a substring of the old one, like this:

```
public class App {
        public static void main(String[] args) {
String s1 = "Hello from Java!";
char c1 = s1.charAt(0);
System.out.println("The first character of \"" + s1 + "\" is " + c1); char chars1[] =
  s1.toCharArray();
System.out.println("The second character of \"" + s1 + "\" is " + chars1[1]); char
  chars2[] = new char[5];
s1.getChars(0, 5, chars2, 0);
System.out.println("The first five characters of \"" + s1 + "\" are " + new
  String(chars2));
String s2 = s1.substring(0, 5);
System.out.println("The first five characters of \"" + s1 + "\" are " + s2); }
}
```

Here's the result of running this program:

```
C:\>java App
The first character of "Hello from Java!" is H
The second character of "Hello from Java!" is e
The first five characters of "Hello from Java!" are Hello
The first five characters of "Hello from Java!" are Hello
```

Searching For and Replacing Strings

You can search strings for characters and substrings by using the `indexOf()` and `lastIndexOf()` methods. The `indexOf()` method returns the zero-based location of the first occurrence in a string of a character or substring, and `lastIndexOf()` returns the location of the last occurrence of a character or substring.

Here's an example that shows how to use `indexOf()` and `lastIndexOf()` methods:

```
public class App {
        public static void main(String[] args) {
String s1 = "We have drawn a nice drawing.";
System.out.println("The first occurrence of \"draw\" is " + "at location " +
  s1.indexOf("draw"));
System.out.println("The last occurrence of \"draw\" is " + "at location " +
  s1.lastIndexOf("draw"));
. . .
```

The `String` class also has a `replace()` method, which lets you replace all occurrences of a single character with another single character. You might think this violates the idea that you can't change the text in a `String` object; however, this method creates an entirely new `String` object. Here's an example showing how this method works (note that we turn all occurrences of the letter *h* into the letter *f* in a text string):

```
public class App {
        public static void main(String[] args) {
String s1 = "we have drawn a nice drawing";
System.out.println("The first occurrence of \"draw\" is " + "at location " +
  s1.indexOf("draw"));
System.out.println("The last occurrence of \"draw\" is " + "at location " +
  s1.lastIndexOf("draw"));
String s2 = "Edna, you're hired!";
System.out.println(s2.replace('h', 'f')); }
}
```

Here's the result of this code:

```
C:\>java App
The first occurrence of "draw" is at location 8
The last occurrence of "draw" is at location 21
Edna, you're fired!
```

Changing Case in Strings

The Novice Programmer says, "The Big Boss told me my program's output wasn't emphatic enough. Do you have any ideas?", "Try the `toUpperCase()` method", you say.

You can use the `toLowerCase()` method to convert a string into lowercase, and you can use the `toUpperCase()` method to convert it into uppercase.

Here's how this looks in code:

```
public class App {
        public static void main(String[] args) {
System.out.println("Hello from Java!".toLowerCase());
System.out.println("Hello from Java!".toUpperCase());     }
}
```

Here's the result of this program:

```
C:\>java App
hello from java!
HELLO FROM JAVA!
```

Checking for Empty String

Java SE 6 onwards, you can check for empty string by using a new method `isEmpty()` which has been introduced in `String` class. The `isEmpty()` method returns true if and only if the length of string is 0. Here's an example:

```
public class App {
  public static void main(String[] args) {
String str="";
System.out.println("The string is Empty?-"+str.isEmpty()); }
}
```

Here's the result:

```
C:\>java App
The string is Empty?-true
```

Formatting Numbers in Strings

You can format numbers in strings by using the `NumberFormat` class of the `java.text` package. This class supports the `format()`, `setMinimumIntegerDigits()`, `setMinimumFractionDigits()`, `setMaximumIntegerDigits()`, and `setMaximumFractionDigits()` methods. Here's an example, using the `setMaximumFractionDigits()` method, in which we round off a `double` value as we format it:

```
import java.text.*;
public class App {
        public static void main(String[] args) {
double value = 1.23456789;
NumberFormat nf = NumberFormat.getNumberInstance();
nf.setMaximumFractionDigits(6);
String s = nf.format(value);
System.out.println(s); }
}
```

Here's the result:

```
C:\>java App
1.234568
```

The **StringBuffer** Class

"Hmm", says the Novice Programmer. "I have stored my whole novel in a `String` object, but now I can't change it. What's wrong?" "You can't change the text in a `String` object", you say. "You have to use a `StringBuffer` object instead". "Now you tell me", replies the NP.

The `StringBuffer` class gives you much of what the `String` class offers—and something more: The ability to modify the actual string. Here's an example that uses the `StringBuffer` class `replace` method to change the contents of a `StringBuffer` object from "Hello from Java!" to "Hello to Java!":

```
public class App {
        public static void main(String[] args) {
StringBuffer s1 = new StringBuffer("Hello from Java!");
s1.replace(6, 10, "to");
System.out.println(s1);   }
}
```

Here's the result of running this code:

```
C:\>java App
Hello to Java!
```

You'll find the `StringBuffer` class constructors in Table 2.5 and its methods in Table 2.6. We'll put the `StringBuffer` class to work in the next few topics.

Table 2.5: StringBuffer class constructors

Constructors	Does this
`StringBuffer()`	It constructs a string buffer with no characters in it and a capacity of 16 characters
`StringBuffer(CharSequence seq)`	It constructs a string buffer that contains the same characters as the specified `CharSequence`
`StringBuffer(int capacity)`	It constructs a string buffer with no characters in it and the given capacity
`StringBuffer(String str)`	It constructs a string buffer so that it represents the same sequence of characters as the argument string

Table 2.6: StringBuffer class methods

Method	Does this
`StringBuffer append (boolean b)`	It appends the string representation of the `boolean` argument to the sequence
`StringBuffer append(char c)`	It appends the string representation of the `char` argument to the sequence
`StringBuffer append (char[] str)`	It appends the string representation of the `char` array argument to the sequence
`StringBuffer append(char[] str, int offset, int len)`	It appends the string representation of a subarray of the `char` array argument to the sequence
`StringBuffer append(CharSequence s)`	It appends the specified `CharSequence` to the sequence
`StringBuffer append(CharSequence s, int start, int end)`	It appends a subsequence of the specified `CharSequence` to the sequence
`StringBuffer append (double d)`	It appends the string representation of the `double` argument to the sequence

Table 2.6: StringBuffer class methods

Method	Does this
`StringBuffer append(float f)`	It appends the string representation of the `float` argument to the sequence
`StringBuffer append(int i)`	It appends the string representation of the `int` argument to the sequence
`StringBuffer append(long l)`	It appends the string representation of the `long` argument to the sequence
`StringBuffer append (Object obj)`	It appends the string representation of the `Object` argument to the sequence
`StringBuffer append (String str)`	It appends the string to this character sequence
`StringBuffer append (StringBuffer sb)`	It appends the given `StringBuffer` to the sequence
`String appendCodePoint(int codePoint)`	It appends the string representation of the `codePoint` argument to the sequence
`int capacity()`	It yields the current capacity of the string buffer
`char charAt(int index)`	It yields the given character of the sequence represented by the string buffer, as indicated by the `index` argument.
`int codePointAt(int index)`	It yields the character (Unicode code point) at the specified index.
`int codePointBefore(int index)`	It yields the character (Unicode code point) before the specified index.
`int codePointCount(int beginIndex, int endIndex)`	It yields the number of Unicode code points in the range of given index arguments.
`StringBuffer delete(int start, int end)`	It removes characters in a substring of the sequence.
`StringBuffer deleteCharAt(int index)`	It removes the character at the given position in the sequence.
`void ensureCapacity(int minimumCapacity)`	It ensures that the capacity of the buffer is at least equal to the given minimum.
`void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`	It copies characters from the sequence into the destination character array `dst`.
`int indexOf(String str)`	It returns the index of first occurrence of the given substring.
`int indexOf(String str, int fromIndex)`	It returns the index of first occurrence of the given substring, starting at the given `fromIndex`.
`StringBuffer insert(int offset, boolean b)`	It inserts the string representation of the `boolean` argument into the sequence.
`StringBuffer insert(int offset, char c)`	It inserts the string representation of the `char` argument into the sequence.
`StringBuffer insert(int offset, char[] str)`	It inserts the string representation of the `char` array argument into the sequence.
`StringBuffer insert(int index, char[] str, int offset, int len)`	It inserts the string representation of a subarray of the `str` array argument into the sequence.
`StringBuffer insert(int dstOffset, CharSequence s)`	It inserts the given `CharSequence` into the sequence.
`StringBuffer insert(int dstOffset, CharSequence s, int start, int end)`	It inserts a subsequence of the given `CharSequence` into the sequence.
`StringBuffer insert(int offset, double d)`	It inserts the string representation of the `double` argument into the sequence.
`StringBuffer insert(int offset, float f)`	It inserts the string representation of the `float` argument into the sequence.

Table 2.6: StringBuffer class methods

Method	Does this
`StringBuffer insert(int offset, int i)`	It inserts the string representation of the second `int` argument into the sequence.
`StringBuffer insert(int offset, long l)`	It inserts the string representation of the `long` argument into the sequence.
`StringBuffer insert(int offset, Object obj)`	It inserts the string representation of the `Object` argument into this sequence.
`StringBuffer insert(int offset, String str)`	It inserts the string into the sequence.
`StringBuffer lastIndexOf(String str)`	It yields the index of the rightmost occurrence of the specified substring.
`StringBuffer lastIndexOf(String str, int fromIndex)`	It yields the index of the last occurrence of the specified substring.
`int length()`	It yields the length (in characters).
`int offsetByCodePoints(int index, int codePointOffset)`	It yields the index within this sequence that is offset from the given `index` by the `codePointOffset` code points.
`StringBuffer replace(int start, int end, String str)`	It replaces the characters in a substring of this sequence with the characters in the given string.
`StringBuffer reverse()`	It replaces the character sequence contained in this sequence with the reverse of the sequence.
`void setCharAt(int index, char ch)`	It sets the character at the given index of the sequence to `ch`.
`void setLength(int newLength)`	It sets the length of the character sequence.
`CharSequence subSequence(int start, int end)`	It yields a new sequence of characters, which is a subset of this sequence.
`String substring(int start)`	It yields a new string that contains a subsequence of characters currently contained in this sequence. The substring begins at the given index.
`String substring(int start, int end)`	It yields a new string that contains a subsequence of characters currently contained in this sequence.
`String toString()`	It yields a string representing the data in this sequence.
`Void trimToSize()`	It tries to reduce the storage space used for the sequence.

Creating StringBuffers

You can create `StringBuffer` objects using the `StringBuffer` class constructors. For example, here's how to create an empty `StringBuffer` object (which is set up with space for 16 characters, by default) and then insert some text into it:

```
public class App { public static void main(String[] args) {
StringBuffer s1 = new StringBuffer();
s1.insert(0, "Hello from Java!");
System.out.println(s1);
. . .
```

Here's how to initialize a new `StringBuffer` object with a string:

```
public class App {
        public static void main(String[] args) {
StringBuffer s1 = new StringBuffer();
s1.insert(0, "Hello from Java!");
System.out.println(s1);
StringBuffer s2 = new StringBuffer("Hello from Java!");
```

```
    System.out.println(s2);
    . . .
```

You can also create a `StringBuffer` object with a specific length, like this:

```
public class App {
        public static void main(String[] args) {
StringBuffer s1 = new StringBuffer();
s1.insert(0, "Hello from Java!");
System.out.println(s1);
StringBuffer s2 = new StringBuffer("Hello from Java!");
System.out.println(s2);
StringBuffer s3 = new StringBuffer();
s3.insert(0, "Hello from Java!");
System.out.println(s3);
}
}
```

Here's the result of this code:

```
C:\>java App
Hello from Java!
Hello from Java!
Hello from Java!
```

Getting and Setting StringBuffer Lengths and Capacities

The `StringBuffer` class provides the `length()` method to find the length of the text in the `StringBuffer` objects. Apart from this, the class provides the `capacity()` method to identify the amount of memory space allocated for the text. You can also set the length of the text in a `StringBuffer` object with the `setLength()` method, which lets you truncate strings or extend them with null characters (that is, characters whose Unicode codes are 0).

Let's create a program to identify the string length and capacity. We will also change the length of the string:

```
public class App {
        public static void main(String[] args) {
StringBuffer s1 = new StringBuffer("Hello from Java!");
System.out.println("The length is " + s1.length());
System.out.println("The allocated length is " + s1.capacity());
s1.setLength(2000);
System.out.println("The new length is " + s1.length()); }
}
```

Here's what this program looks like when it's run:

```
C:\>java App
The length is 16
The allocated length is 32
The new length is 2000
```

Setting Characters in String Buffers

"Help!" the Novice Programmer cries, "I need to change some text in my novel!" "You can try the `setCharAt()` method", you say helpfully.

To read characters in a `StringBuffer` object, you can use the `charAt()` and `getChars()` methods, just as you can with `String` objects. However, in `StringBuffer` objects, you can also set individual characters by using the `setCharAt()` method.

Here's an example in which we change the text: "She had a wild look in her eyes" to "She had a mild look in her eyes" by using `setCharAt()`:

```
public class App {
```

```
        public static void main(String[] args) {
StringBuffer s1 = new
StringBuffer("She had a wild look in her eyes");
s1.setCharAt(10, 'm');
System.out.println(s1); }
}
```

Here's the result:

```
C:\>java App
She had a mild look in her eyes
```

Appending and Inserting Using StringBuffers

"The `setCharAt()` method doesn't do it for me", the Novice Programmer says. "I really need some way of editing the text in `StringBuffer` objects as a string, not as individual characters". "OK", you say. "Use the `append()` and `insert()` methods". You can use the `append()` method to append strings to the text in a `StringBuffer` object, and you can use the `insert()` method to insert text at a particular location.

Here's an example that starts with the text "Hello", appends " Java!", and then inserts "from" into the middle of the text, using `append()` and `insert()` methods:

```
public class App {
        public static void main(String[] args) {
StringBuffer s1 = new StringBuffer("Hello");
s1.append(" Java!");
System.out.println(s1);
s1.insert(6, "from ");
System.out.println(s1); }
}
```

Here's what this code produces:

```
C:\>java App
Hello Java!
Hello from Java!
```

Deleting Text in StringBuffers

You can delete text in a `StringBuffer` object by using the `delete()` and `deleteCharAt()` methods. For example, here's how to change the text "We are not having a good time." to "We are having a good time." with `delete()` method. (To use this method, you just specify the range of the characters you want to delete):

```
public class App {
        public static void main(String[] args) {
StringBuffer s1 = new
StringBuffer("We are not having a good time.");
s1.delete(7, 11);
System.out.println(s1); }
}
```

Here's the result:

```
C:\>java App
We are having a good time.
```

Replacing Text in StringBuffer

"I'm writing a text editor by using the `StringBuffer` class", the Novice Programmer says, "but there's one thing I can't figure out—how can I replace text with other text? Do I have to delete it and then insert the new text?" "No", you answer. "It's simple—just use the `replace()` method".

In fact, you've already seen how to use `replace()` method; you just specify a character range and the new text that should replace that range, like this:

```
public class App {
        public static void main(String[] args) {
StringBuffer s1 = new StringBuffer("Hello from Java!");
s1.replace(6, 10, "to");
System.out.println(s1); }
}
```

Here's the result of the preceding code:

```
C:\>java App
Hello to Java!
```

Using the **Wrapper** Class

The Novice Programmer is confused and says, "I'm unable to put primitive values into a collection" "Ok", you say. "That can be solved easily, I'll tell you how".

Java uses primitive data types such as `int`, `double`, `char` which does not come under the hierarchy of objects. These data types cannot be called by reference. There could be a situation in which we need to use the primitive data types as objects. Java provides a `wrapper` class in order to use the primitive data types as objects, which wraps the primitive data type within an object. So, every primitive data type has a corresponding class defined in the Java API class library.

Table 2.7 illustrates the primitive data type and the corresponding `wrapper` class:

Table 2.7: Wrapper classes for the primitive types

Primitive type	Wrapper class
int	Integer
short	Short
long	Long
byte	Byte
float	Float
double	Double
char	Character
boolean	Boolean

There are different methods defined in order to return the value of an object. For example, `byteValue()` method returns the value of an object as `byte`; similarly, `doubleValue()` method will return the values as `double`. The following example is giving you the demonstration of using the `wrapper` class:

```
public class App {
  public static void main (String args[]) {
        Integer intwrap = new Integer(250);
        Character chrwrap = new Character('H');
        int int1 = intwrap.intValue();
        char chr1= chrwrap.charValue();
        System.out.println(int1 + "    " + intwrap);
        System.out.println(chr1+"    "+ chrwrap); }
}
```

This program compiles and upon executing shows the result as:

```
C:\>java App
250    250
H    H
```

The following lines from the preceding program are *encapsulating* a value within an object. This feature is known as *boxing:*

```
Integer intwrap = new Integer(250);
Character chrwrap = new Character('H');
```

The following lines of code are extracting the values from the type wrapper. This feature is known as *unboxing:*

```
int int1 = intwrap.intValue();
char chr1= chrwrap.charValue();
```

Autoboxing and Unboxing of Primitive Types

As a Java programmer, you probably know that you cannot put an `int` (or other primitive values) in a collection (Refer Chapter 21). As only object references can be held by a collection, you are required to *box* primitive values into the corresponding `wrapper` class (for example, `Integer` for `int`). If you need to take out an object from a collection, you get the `Integer` value that you had put in earlier. As in the case of `wrapper` class, there is a need to define the object by using the `new` keyword (which allocates the memory for the variable). If this boxing and unboxing needs to be done manually every time, it would complicate your code, therefore, making it difficult to write and understand. Here, the autoboxing and unboxing comes in and is done automatically.

Let's consider the following example which demonstrates the feature of autoboxing and unboxing. Here's the whole program, which is `App.java`:

```
public class App {
   public static void main(String args[]) {
Integer intbox1,intbox2,intbox3;
intbox1 =250;
int a= intbox1;
System.out.println("The value of intbox1 is "+ intbox1);
System.out.println("The value of int1 is "+ a);
intbox2=500;
System.out.println("value of intbox2 is "+ intbox2);
intbox2=intbox2+1;
System.out.println("value of intbox2 after increment is " + intbox2);
   intbox3= intbox2+(intbox2/5);
   System.out.println("value of intbox 3 after evaluation is "+ intbox3);
         Boolean bbox1= true;
                 if (bbox1)
System.out.println ("bbox1 contains the true
value");
         Character chbox1='H';
         char chbox2= chbox1;
         System.out.println("value of chbox2 is "+ chbox2);
         Integer intbox4 = Autobox2.mthdbox(500);
         System.out.println("The value obtained from the method is "+ intbox4); }
}
class Autobox2 {
   static int mthdbox(Integer b) { return b; }
}
```

The output of the program will be:

```
C:\> java App
The value of intbox1 is 250
The value of int1 is 250
value of intbox2 is 500
value of intbox2 after increment is 501
value of intbox3 after evaluation is 601
bbox1 contains the true value
value of chbox2 is H
The value obtained from the method is 500
```

Let's discuss what is happening in the preceding program, from the previous code we have:

```
public class App {
  public static void main(String args[]) {
        Integer intbox1,intbox2,intbox3;
        intbox1 =250;
        int a= intbox1;
        System.out.println("The value of intbox1 is "+ intbox1);
        System.out.println("The value of a is "+ a);
  . . . }
  }
```

The following code line in the preceding program is encapsulating a value within an object:

```
intbox1 =250;
```

This feature is known as *autoboxing* as we have not defined the object explicitly by using the `new` keyword. Again, the following line is shown as extracting the values. This feature is known as *unboxing*:

```
int a= intbox1;
```

If we'll print the results of the two variables defined, it will yield the same result as assigned. If you need an `int`, it is required that you *unbox* the `Integer` by using the `intValue()` method.

Let's take another section of the code from the program:

```
public class App {
  public static void main(String args[]) {
        Integer intbox1,intbox2,intbox3;
        intbox1 =250;
        int a= intbox1;
        System.out.println("The value of intbox1 is "+ intbox1);
        System.out.println("The value of a is "+ a);
        intbox2=500;
        System.out.println("value of intbox2 is "+ intbox2);
        intbox2=intbox2+1;
        System.out.println("value of intbox2 after increment is " + intbox2);
        intbox3= intbox2+(intbox2/5);
        System.out.println(" value of intbox 3 after evaluation is "+ intbox3);
        . . . }
}
```

In these lines, the expression is being calculated and the result obtained will again be *reboxed*. In the line, `intbox2=500`, autoboxing is taking place.

Note the expression `intbox2=intbox2+1`. It will automatically unbox the variable and then result back into the variable by reboxing. You'll get the result of different variables used in the expression. It will display the values before evaluating and after evaluating the expression. You can see this later in this section. Let's take some more lines of the code:

```
public class App {
  public static void main(String args[]) {
        Integer intbox1,intbox2,intbox3;
        intbox1 =250;
        int a= intbox1;
        System.out.println("The value of intbox1 is "+ intbox1);
        System.out.println("The value of a is "+ a);
        intbox2=500;
        System.out.println("value of intbox2 is "+ intbox2);
        intbox2=intbox2+1;
        System.out.println("value of intbox2 after increment is " + intbox2);
        intbox3= intbox2+(intbox2/5);
        System.out.println("value of intbox 3 after evaluation is "+ intbox3);
        Boolean bbox1= true;
        if (bbox1)
              System.out.println ("bbox1 contains the true value");
              Character chbox1='H';
        char chbox2= chbox1;
```

```
            System.out.println("value of chbox2 is "+ chbox2);
      . . . }
   }
```

You can take the `boolean` and `char` variables also. The auto-unboxing takes place in the `if` condition when the condition expression gets evaluated. Similarly, the line `Character chbox1='H'` autoboxes the char and `char chbox2= chbox1` auto-unboxes the `char`.

At times, situations are such that you may require passing values of primitive type to a method that requires the argument to be referenced to an object. In such cases, the compiler will automatically convert those primitive values to the required class types where permissible. Let's take more lines of code:

```
   public class App {
     public static void main(String args[]) {
           Integer intbox1,intbox2,intbox3;
           intbox1 =250;
           int a1= intbox1;
           System.out.println("The value of intbox1 is "+ intbox1);
           System.out.println("The value of a1 is "+ a1);
           intbox2=500;
           System.out.println("value of intbox2 is "+ intbox2);
           intbox2=intbox2+1;
           System.out.println("value of intbox2 after increment is " + intbox2);
           intbox3= intbox2+(intbox2/5);
           System.out.println("value of intbox 3 after evaluation is "+ intbox3);
           Boolean bbox1= true;
           if (bbox1)
                   System.out.println ("bbox1 contains the true value");
           Character chbox1='H';
           char chbox2= chbox1;
           System.out.println("value of chbox2 is "+ chbox2);
           Integer intbox4 = Autobox2.mthdbox(500);
           System.out.println("The value obtained from the method is "+ intbox4);
    }
   }
   class Autobox2 {
           static int mthdbox(Integer b1) { return b1; }
   }
```

Here we are defining a method `mthdbox()` with `Integer` type parameter value `b1` is being passed, which will return the `int` value in class `Autobox2`. So, in order to call the method `mthdbox()`, note the expression `Integer intbox4 = Autobox2.mthdbox(500)` defined in the `App` class. The value can only be called by explicitly mentioning the class of the `function` from which it is called. So, the value of `b1` will be displayed as a result. The `return` statement will auto-unbox the expression.

Alternately, you can perform the autoboxing in methods by using another way as shown here:

```
   class App {
           static int mthdbox(Integer b1) { return b1; }
     public static void main(String args[]) {
           Integer intbox4 = mthdbox(500);
           System.out.println(intbox4); }
   }
```

Here, there is no need to explicitly call the function by using the class name as it itself contains the `main` function.

Learning the Fundamentals of Varargs Methods

The NP is tensed, "Is there an easy way by which we could avoid using arrays and still format a message"? "It's simple" you say, "use varargs". "What's that?" asks NP.

Varargs means variable-length arguments or a method that takes a variable number of arguments. This feature was introduced in Java SE 5. This makes it easier to create methods whose argument list widely varies.

To understand this better, we can compare this with `arrays`. If you needed to create a method that could take an arbitrary number of values, you could put the values into the array prior to invoking the method.

For example, shown here is the `MessageFormat` class that formats a message:

```
    Object[] arguments = {
          new Integer(7), new Date(), "a disturbance in the Force" };
    String result = MessageFormat.format("Nature caused {2} , on {1,date}
    {1,time}" + "{0,number,integer}"., arguments);
```

Basically, instead of passing multiple arguments in an array, varargs automates and hides the process. Additionally, it is upwardly well matched with the existing APIs. For example, the `MessageFormat.format()` method now has this declaration:

```
public static String format(String pattern, Object... arguments);
```

The three periods after Object indicate that you can pass an array or a sequence of arguments as final argument. However, it must be noted that varargs can be used only in the final argument position.

Varargs can be useful in core APIs including reflection, message formatting, and the new `print` facility. If you are an API designer, you should use them in moderation—only when the benefit is truly undeniable. Care must be taken so that the varargs method is not overloaded or it will be difficult for you to figure out which overloaded method gets called; but you can take advantage of them whenever the API offers them. Let's now understand the concept of varags through a small example given here:

```
public class App {
  static void mthdvar(int ...int1) {
        System.out.print("Total number of arguments passed are: "
              +int1.length+ " Contents: ");
        for (int int2:int1) System.out.print(int2 +"   ");
        System.out.println(); }
  public static void main (String args[]) {
        mthdvar();
        mthdvar(100);
        mthdvar(200,300,400); }
}
```

The result of the preceding code is as follows:

```
Total number of arguments passed are: 0 Contents:
Total number of arguments passed are: 1 Contents: 100
Total number of arguments passed are: 3 Contents: 200   300   400
```

So, in the preceding examples, we have defined a method `mthdvar()`, in which `int1` is operated as an `array`. The three dots used in the line will be known by the compiler as an `array`. As you can see in the `main()`, method, `mthdvar()` has been called by using three different arguments; in the first case, we have not passed any arguments.

The contents of this array arguments can be fetched by using the `for-each` statement. Let's briefly go through the `for-each` loop. You will be explained about the `for-each` loop as well as other varieties of loops in Chapter 3.

TIP

The statement for (type any-var : collections) is in general form. The word type is for the data type which should be the same as defined in the array or passed in the argument list. Collection is being considered as the type of objects whether array or some list value specifies one at a time from first element to the last element. Let's take a small example to understand this concept.

```
public class App {
  public static void main (String args[]) {
        int list1[]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
        int total=0;
        for (int int1:list1) {
              System.out.println("Values of element is :" +int1);
              total=total+int1; }
System.out.println("Total sum of all the values defined  is: "+ total); }
}
```

Its output is displayed as follows:

```
Values of element is :10
Values of element is :20
```

```
Values of element is :30
Values of element is :40
Values of element is :50
Values of element is :60
Values of element is :70
Values of element is :80
Values of element is :90
Values of element is :100
Total sum of all the values defined is: 550
```

Overloading Varargs Methods

The overloading feature of varargs allows a set of functions that perform a similar operation to be collected under a common mnemonic name, such as `print()`. This removes the lexical intricacy of providing each function with a unique name. Alternatively, depending on the particular data in use, the function overloading is used when you have a single task, which has two or more implementations or algorithms. By overloading the function, you can hide other available algorithms from the user.

Overloading methods should be dealt with very carefully. The compiler decides which version of an overloaded method will be called based on declared compile-time type, not runtime type. In the case in which overloaded methods have the same number of arguments, the rules regarding this decision can sometimes be a bit tricky. If there is ambiguity, the design can be simplified by:

- using different method names and avoiding overall overloading.
- retaining overloading, but ensuring each method has a distinct number of arguments.

In addition, it is recommended that varargs should not be used when a method is overloaded, since this makes it harder to determine which method is being called.

NOTE

Overloading requires methods with distinct signatures. The signature of a method includes its name and the ordered list of its argument types. All other items appearing in a method header, such as exceptions, return type, final, and synchronized, do not contribute to a method's signature.

Here's an example to understand the concept:

```
public class App {
   public static void Averg(Integer... List) {
          Integer total=0; int ListCount=0;
          for(Integer cnt:List) {
                  total+=cnt;
                  ListCount++; }
          System.out.println((double)total/ListCount);
          System.out.println((int)total/ListCount); }
   public static void main(String args[]) {
          Averg(7); Averg(1,2,3);
          Averg(); }
}
```

Here's the result:

```
C:\> java App
7.0
7
2.0
2
NaN
Exception in thread "main" java.lang.ArithmeticException: / by zero
   at App.Averg(App.java:8)
   at App.main(App.java:11)
```

In this scenario, we need to be careful that the implementation handles an empty vararg list correctly. Here we could have avoided the issue by adding a runtime check to make sure that the `List` was not empty, even better would be to make the first integer necessary so that the last call to `Averg` would not generate a type mismatch error. The last call to `Averg` also generates `NaN`, Not a Number, as it is the result of 0/0.

Similar type of elements must be passed to the varargs list, although that type can be of any type or a user-defined type.

Trying to combine varargs with optional parameters is not a good idea—though it just doesn't work most of the time (type mismatch errors are common). The resulting functionality generally isn't very useful even when the code compiles without errors.

Also, once you are inside the function body, your varargs parameter is a list, completely impossible to tell apart from a list passed as a parameter. This means that passing a variable number of arguments through one function to another could be really tough.

```
Test( varargs List ) { Counter( List ); }
Counter ( varargs List ) { print( ListCount(List) ); }
main () { Test( 1, 2, 3) ; }
```

Here's an example to understand the concept:

```
public class App {
  public static void Test(Integer... List) { counter(List); }
  public static void counter(Integer... List) {
        int ListCount=0;
        for(Integer cnt:List) { ListCount++;    }
        System.out.println(ListCount); }
  public static void main(String args[]) { Test(1,2,3); }
}
```

Here's the result:

```
C:\>java App
3
```

Fortunately, this issue doesn't come up too often.

Learning the Ambiguity in Varargs Methods

However, varargs has its own advantages, but this has not been very popular with the developer community. It is possible that the programmers may seem to be comfortable with the earlier solutions and are not so with this feature. Also, there may be some unknown problems with this new technology. To have a better view of this, we can proceed by putting various examples and scenarios as follows.

When the compiler creates code to call the `varargs` method, it may create an instance of a special class for a chosen small number of arguments. It is obvious that a special implementation of *Indexed for one element* will always be faster on modern `HotSpot` than a single-element array (although natïve iteration with `for-each` facility will require extra `Iterator` object, but that can be fixed, if needed, by recognizing Indexed object as a special case in `for-each` statement). Performance will certainly be better for the array-based varargs when the number of arguments is large, but it is a matter of debate that people who really need varargs and drive their inclusion into Java do not care about performance much, and it will be perfectly fine for them to deal with an extra object that implements Indexed interface and wraps underlying array for many argument cases.

The varargs syntax introduces an inconsistency between the behavior of parameters and local variables. At present, the parameter declarations behave exactly like local variables: passing values `a` and `b` to a method `Test(int x, String y)` is directly equivalent to `int x = a; String y = b`. Using varargs, you can pass values `a` and `b` to the method with a single parameter like this: `Trial(Object... x)`. This implies an equivalent syntax for initializing a local variable to multiple values like this: `Object... x = a, b;`. However, such syntax does not exist. The problem is not the use of syntax. It shows that the varargs feature doesn't feel well integrated into the language.

When you use overloaded methods, with varargs it is more difficult to tell which method will get called. If this is combined with autoboxing, it can be even more confusing. We have no idea which `Test` method gets called. However, this problem would not exist with *Concise Object Array Literals*.

To make varargs acceptable, it can be said that methods declared with a "…" parameter type needs to use it as the last argument type (like in C, C++, C#), and require that the method is less precise than any other overloading method declaring more specific argument types (like in C++). But it is difficult when reading the caller code if a more specific method will be called or if the vararg method will be used. This is determined at

compile time by looking at the methods of the callee `class`; but if that `class` is extended later with more specific methods, the old caller will start using them instead of using the vararg method. This is a case where the type signature safety has not been taken care of. Many more examples are available that show the ambiguity in varargs, but we need to find those by getting accustomed to the technology and to popularize the use of the same.

Using Non-Reifiable Formal Parameters

The non-reifiable formal parameter is a type of parameter passed to the methods and classes whose type information is not completely available at runtime. Examples of the non-reifiable types are the parameterized types such as `ArrayList<Integer>` and `List<Double>`. These parameters lose their type information during the type erasure process, which is performed at the compile time. In the type erasure process, the compiler removes all information of the type parameters and type arguments within a class or method that uses generics. For example, after the type erasure process, the `Cat<String>` type is converted to the raw `Cat` type. A raw type is the name of a class or interface without any type information. The process of type erasure is performed to make the Java applications using generics compatible with the application developed before the generics.

The type erasure process has consequences when parameters of a vararg method are of the non-reifiable type. You will understand this better with an example:

```
import java.util.*;
public class App {
  public static void faultyGenericMethod(List<String>... listAdd) {
        Object[] objectCollArray = listAdd;  // Valid
        objectCollArray[0] = Arrays.asList(new Integer(44));
        String s = listAdd[0].get(0); // ClassCastException thrown at this
                                      // line
  }
  public static void main (String args[]){
        List<String> test=new ArrayList<String>(5);
        faultyGenericMethod(test);
  }
}
```

Here, the first statement of the `faultyGenericMethod()` method assigns the formal list non-reifiable parameter `listAdd` to the object array `objectCollArray`. The list parameter can be assigned to the object array because the formal parameters of the Vararg method are converted to arrays during compilation. In our case, the `listAdd` parameter of the `List<String>` type is converted to `List[]`, which can be assigned to the `object [] objectCollArray`. The assignment of the `listAdd` parameter to the `objectCollArray` object array can cause potential heap pollution. The heap pollution is caused because the `Integer` object whose type does not match the parameterized `String` type of the `listAdd` vararg formal parameter can also be assigned to the object array `objectCollArray`. The `objectCollArray` object in turn refers to the `listAdd` formal parameter whose type is `List<String>`. As a result, the compiler does not raise any warning or error when the `List` object of `Integer` type is added to the first element of the `objectCollArray` array, as shown in the following code statement:

```
objectCollArray[0] = Arrays.asList(new Integer(44));
```

At the runtime, when the first element of the `listAdd` formal parameter, whose parameterized type is `String`, is assigned to a String variable s, the `ClassCastException` exception is thrown. The `ClassCastException` exception is thrown because the first element of the `listAdd` formal parameter is of Integer type and is assigned to a String variable s. The retrieval and assignment of the first element of the `listAdd` parameter is shown in the following code statement:

```
String s = listAdd[0].get(0);
```

Here are the exceptions that the above program throws upon compiling and executing:

```
C:\>javac App.java
Note: App.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
C:\>java App
```

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be
  cast to java.lang.String
              at App.faultyGenericMethod<App.java:6>
             at App.main<App.java:11>
```

Summary

In this chapter, we have learned about the use of variables, literals, data types, arrays, strings, and Wrapper classes. After getting the conceptual knowledge on the concepts, the Immediate Solutions section has helped you to declare and initialize variables, create binary literals, and use underscores in numeric literals. You also came to know how to perform operations on arrays and strings. You have learned the use of StringBuffer classes and wrapper classes. The chapter ends with the discussion on autoboxing and unboxing of primitive types, and the fundamentals of varargs methods.

In the next chapter, you will learn about the operators and control structures.



3

Operators, Conditionals, and Loops

If you need an immediate solution to:	*See page:*
Operator Precedence	81
Incrementing and Decrementing (++ and --)	82
Unary NOT (~ And !)	83
Multiplication and Division (* and /)	83
Modulus (%)	83
Addition and Subtraction (+ and -)	84
Shift Operators (>>, >>>, and <<)	84
Relational Operators (>, >=, <, <=, ==, and !=)	85
Bitwise and Bitwise Logical AND, XOR, and OR (&, ^, and /)	85
Logical (&& and ||)	87
The if-then-else Operator (?:)	87
Assignment Operators (= and [operator]=)	88
Using the Math Class	89
Changes in the Math Class	90
Class StrictMath	90
Comparing Strings	91
The if Statement	92
The else Statement	92
Nested if Statement	93
The if-else Ladders	93
The switch Statement	93
Using Strings in switch Statement	95
The while Loop	95
The do-while Loop	96

If you need an immediate solution to:	*See page:*
The for Loop	97
The for-each Loop	99
Supporting for-each in Your Own Class	101
Nested Loops	102
Using the break Statement	103
Using the continue Statement	103

In Depth

In the previous chapter, we took a look at how Java handles data in basic ways. In this chapter, we'll start doing something with that data as we examine the Java operators, conditionals, and loops.

Storing a lot of data in your program is fine; but unless you do something with it, it's not of much use. Using operators, you can manipulate your data—add, subtract, divide, multiply, and more. With conditionals, you can alter a program's flow by testing the values of your data items. Using loops, you can iterate over all data items in a set, such as an array, working with each data item in succession in an easy way. These represent the next step up in programming power from the previous chapter, and we'll discuss all three of these programming topics here.

Operators

Java provides a very easy way to work with data in a program. Like most languages, it provides a number of built-in operators. For example, if one variable contains value 10 and another variable contains 20, the variables can be added using the + operator, as shown in this code:

```
public class App {
        public static void main(String[] args) {
        int operand1 = 20, operand2 = 10, sum;
        sum=operand1+operand2;
        System.out.println(operand1 + " + " +operand2 + " = " + sum); }
}
```

Here's the result of this code:

```
C:\>java App
20 + 10 = 30
```

So, what operators does Java offer? Here's a list of all of them:

- --(decrement)
- - (subtraction)
- ! (logical unary NOT)
- != (not equal to)
- % (modulus)
- %= (modulus assignment)
- & (logical AND)
- && (short-circuit AND)
- &= (bitwise AND assignment)
- * (multiplication)
- *= (multiplication assignment)
- / (division)
- /= (division assignment)
- ?: (ternary if-then-else)
- ^ (logical XOR)
- ^= (bitwise XOR assignment)
- | (logical OR)
- || (short-circuit OR)
- |= (bitwise OR assignment)
- ~ (bitwise unary NOT)
- + (addition)

- **++** (increment)
- **+=** (addition assignment)
- **<** (less than)
- **<<** (shift left)
- **<<=** (shift left assignment)
- **<=** (less than or equal to)
- **=** (assignment)
- **-=** (subtraction assignment)
- **==** (equal to)
- **>** (greater than)
- **>=** (greater than or equal to)
- **>>** (shift right)
- **>>=** (shift right assignment)
- **>>>** (shift right with zero fill)
- **>>>=** (shift right zero fill assignment)

You'll see these operators at work in this chapter. Operators that take one operand are called *unary* operators. Those that take two operands—for example, addition (a + b)—are called *binary* operators. There's even an operator, `?:`, that takes three operands—the *ternary* operator.

NOTE

Besides the built-in operators, we'll also cover the Java `Math` class in this chapter, which allows you to add a lot more math power to your programs, including exponentiation (unlike other languages, Java has no built-in exponentiation operator), logarithms, trigonometric functions, and more. StrictMath classes (you can find more about these classes at http://java.sun.com/javase/8/docs/api/) ensure that the same mathematical precision is used no matter what machine your program is running on.

Conditionals

The next step up from using simple operators is to use conditional statements, also called *branching statements*, in your code. You use conditional statements to make decisions based on the value of your data and make the flow of the program go in different directions accordingly.

Say you wanted to report on the weather; and if it's less than 80 degrees Fahrenheit, you want to print out a message that reads "It's not too hot". You can do this by checking the current temperature with a Java `if` statement that compares the value in the variable `temperature` to 80; and if that value is under 80, it prints out your message:

```
public class App {
  public static void main(String[] args) {
        int temperature = 73;
        if (temperature < 80) { System.out.println("It's not too hot."); }
  }
}
```

The `if` statement tests whether its condition (the part that appears in the parentheses) is true, which in this case is `temperature < 80`. The Java < (less than) relational operator is used to test whether the value in `temperature` is less than 80. Because we have set that value to 73, the `if` statement's condition is true, which means the code in the body of the `if` statement will be executed. Here's the result of this code:

```
C:\>java App
It's not too hot.
```

You can make `if` statements more complex by adding `else` clauses. These clauses must follow the `if` statement and are executed when the `if` statement condition turns out to be false. Here's an example:

```
public class App {
  public static void main(String[] args) {
```

```
            int temperature = 73;
            if (temperature < 80) { System.out.println("It's not too hot!"); }
            else { System.out.println("It's too hot!"); }
      }
    }
```

As you'll see, there are other conditional statements as well, and we'll put them to work in this chapter, giving the Java syntax a thorough workout.

Loops

Loops enable you to execute a set of statements for a number of times. For example, you may want to print your name five times on the console or calculate the sum of the first hundred natural numbers; in such cases, loops are used. The loop execution depends upon a loop condition. The statements inside a loop are executed as long as the loop condition remains true. Generally, the condition involves a loop index that gets incremented or decremented every time the loop executes.

The general syntax of a `for` loop is as follows:

```
for (initialization_expression; test_conditon; iteration_expression) {
        statement;
}
```

Note that the statement that makes up the body of the `for` loop can be a compound statement, which means it can be made up of several single statements enclosed within curly braces.

You can initialize the loop index with initialization expression, provide a test condition `for` loop termination, and modify loop index value in iteration expression. Note that we are declaring and initializing the loop index at the same time. Isn't it similar to C++? Yes, C++ also allows you to simultaneously declare and initialize the loop index. A simple example of a `for` loop which prints `Kogent` five times is as follows:

```
public class App {
        public static void main(String[] args) {
        for (int loop_index = 1; loop_index < 6; loop_index++){
        System.out.println("Kogent");  }
        }
  }
```

This code prints `Kogent`, using `loop_index`, which starts at 1 and is steadily incremented each time through the loop by 1 till the loop test condition evaluates true. The result of this code is:

```
C:\>java App
Kogent
Kogent
Kogent
Kogent
Kogent
```

There are statements that cause an unconditional jump to other statement in the code. Such statements are called jump statements. These are mainly used to interrupt the control flow within the switch statements and loops. There are three jump statements supported by Java: `break`, `continue`, and `return`. Generally, the `break` statement allows exit from a loop or switch before the execution completes, and the `continue` statement takes control to the beginning of a loop or switch. For the `return` statement, please see the topic "Returning Values from Methods" in Chapter 4.

Here's a simple example of the `break` statement to make things clearer. In this case, the loop index is initialized with value 1 and is incremented by 1 for every execution of the loop. In the code, the test condition states that the loop should execute till the loop index is less than 7:

```
public class App {
      public static void main(String[] args) {
              for (int loop_index = 1; loop_index < 7; loop_index++){
      if(loop_index==5) break;
      System.out.println(loop_index); }
                      System.out.println("Kogent");  }
  }
```

This code prints the value of `loop index` until it becomes 5. Then the `break` statement takes control out of the loop and `Kogent` gets printed. The result of this code is:

```
C:\>java App
1
2
3
4
Kogent
```

As you can see, `for` loop is a powerful one; in fact, it's just one of the many topics coming up in the "Immediate Solutions" section. It's now time to start by using operators, conditional statements, and loops.

Immediate Solutions

Operator Precedence

"Hey," says the Novice Programmer, "Java's gone all wacky again. I tried adding 6 and 24 and then dividing the result by 6. The answer should have been 5, but Java said it's 10". "Probably an operator precedence problem," you say. "Let me check your code."

When a single Java statement contains a number of operators, you may also face problem similar to what the Novice Programmer is facing here. An important concern is which operator does Java execute first? For example, take a look at the Novice Programmer's code in which he tries to add 6 and 24 and then divide the sum by 6:

```
public class App {
        public static void main(String[] args) {
        double value;
        value = 6 + 24/6;
        System.out.println("Value is " + value); }
}
```

The actual result of this code is:

```
C:\>java App
Value is 10.0
```

The Novice Programmer was right; it's different from what he expected. Actually, it's the operator precedence that determines which operator is to be executed first, which second, and so on. Operator precedence rule states that the operators are executed in the order of highest to lowest precedence. The Novice Programmer is getting 10 as output because the / operator has higher precedence than the + operator. Therefore, 24 is firstly divided by 6 and then the result is added to 6, which produces 10.

You can use parentheses to specify Java the way you want the operators to execute. Here's how the Novice Programmer's code would now look; see the parentheses around "6 + 24," which have been used to make sure that this operation is performed first:

```
public class App {
        public static void main(String[] args) {
        double value;
        value = (6 + 24)/6;
        System.out.println("Value is " + value); }
}
```

The result of this code is:

```
C:\>java App
Value is 5.0
```

Table 3.1 spells out the Java operator precedence, from highest to lowest (operators on the same line have the same precedence). Note that at the very highest level of precedence, you'll find `( )`, `[ ]` (the array operator, which you use to get data items at a specific index in an array), and `.` (the dot operator, which you use to specify methods and data members of objects). This means that, for example, you can always use parentheses to set the execution order of operations in Java statements.

We'll go over all these Java operators, in order of precedence, in this chapter, starting with the incrementing and decrementing operators: ++ and --.

Table 3.1: Operator precedence

Operators			
( )	[ ]	.	
++	--	~	!
*	/	%	
+	-		

Table 3.1: Operator precedence			
Operators			
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	[operator]=		

Incrementing and Decrementing (++ and --)

The Programming Correctness Czar appears and says, "C++ has an incrementing operator and a decrementing operator. Does Java support these?" "Sure thing," you say.

The ++ operator increments its operand by 1, and the -- operator decrements its operand by 1. For example, if `value` holds 10 after you execute `value++`, `value` will hold 11.

The ++ and -- operators were introduced in C to make incrementing and decrementing values, which are very common and easy operations. In fact, they were so popular that the incrementing operator was used in C++'s name, indicating that C++ is an incremented version of C.

Here's an important point to note: ++ and -- can be used as either *postfix* operators (for example, `value++`) or *prefix* operators (for example, `++value`). Prefix operators are executed *before* the rest of the statement, whereas postfix operators are executed *after* the rest of the statement. For example, consider the following code:

```
value2 = value1++;
```

The preceding statement post increments `value1`. Note that `value2` will still contain its original value and `value1` gets incremented; the case would have been different if it was a pre-increment operator. Here's an example to make it clear:

```
public class App {
   public static void main(String[] args) {
   int value1 = 10, value2 = 10;
   System.out.println("value1 = " + value1);
   System.out.println("value2 = " + value2);
   value2 = value1++;
   System.out.println("After value2 = value1++");
   System.out.println("value1 = " + value1);
   System.out.println("value2 = " + value2);
   value2 = ++value1;
   System.out.println("Now, after value2 = ++value1");
   System.out.println("value1 = " + value1);
   System.out.println("value2 = " + value2); }
}
```

The result of this code is:

```
C:\>java App
value1 = 10
value2 = 10
After value2 = value1++
value1 = 11
value2 = 10
Now, after value2 = ++value1
```

```
value1 = 12
value2 = 12
```

Unary NOT (~ And !)

The ~ operator is the bitwise unary `NOT` operator, and `!` is the logical `unary NOT` operator. The ~ operator flips all the bits of numeric arguments, and the `!` operator flips true values to false and false values to true. Here's an example in which we flip all the bits of the most positive short value, 32767, to find the most negative short value, and we also flip a Boolean value from true to false:

```
public class App {
        public static void main(String[] args) {
        short s1 = 32767;
        boolean boolean1 = true;
        System.out.println("Most negative short = " + ~s1);
        System.out.println("!true = " + !boolean1); }
}
```

Here's the result of this code:

```
C:\>java App
Most negative short = -32768
!true = false
```

If we had set `s1` to 0 and then flipped its bits with the ~ operator to 1111111111111111 in binary, Java would have displayed the resulting value as `-1`, because it uses two's complement notation for negative numbers. This means the leading bit is 1 for negative numbers and 0 for zero and positive numbers.

TIP

So, why are 1111111111111111 binary equal to -1 in a short variable? If you add it to 1, you end up with 10000000000000000 binary, a number too large for a 16-bit short, so the leading 1 is lost and you end up with 0 (in other words, -1 + 1 = 0).

Multiplication and Division (* and /)

The Programming Correctness Czar says, "I expect that Java has multiplication operator and division operators, just like C++". "Sure," you say.

You use * to multiply values and / to divide values in Java. Here's an example in which we use * and / on double values, and then we do the same thing on integer values. We perform multiplication and division on integer values to show that the fractional part of math results is truncated when you use integers. Therefore, if you want to perform a division operation and still retain precision, you probably shouldn't be using integers. Here's the code:

```
public class App {
        public static void main(String[] args) {
        double double1 = 4, double2 = 6, double3 = 5, doubleResult;
        doubleResult = double1 * double2 / double3;
        System.out.println("4 * 6 / 5 = " + doubleResult);
        int int1 = 4, int2 = 6, int3 = 5, intResult;
        intResult = int1 * int2 / int3;
        System.out.println("With integer math, 4*6/5 = " + intResult); }
  }
```

Here's the result of this code:

```
C:\>java App
4 * 6 / 5 = 4.8
With integer math, 4 * 6 / 5 = 4
```

Modulus (%)

Modulus operator returns the remainder of a division operation. For example, 11%3 returns 2 and 7%2 returns 1 as output. Note that the modulus operator is especially useful when converting between bases because you can use it to successively strip digits off a number by using the modulus operator with the base you are converting into. To see how this works, take a look at the topic "The `while` Loop," later in this chapter. It contains a full example.

Addition and Subtraction (+ and –)

"The multiplication operator is an asterisk and the division operator is a forward slash", the Novice Programmer says, "but those aren't the symbols I learned for these operations in school. What does Java use for plus and minus?" "The usual symbols for plus and minus", you reply.

The old standby numeric operators are + and –, which you use for addition and subtraction respectively. Here's an example:

```
public class App {
        public static void main(String[] args) {
        int operand1 = 5, operand2 = 4, sum, diff;
        sum = operand1 + operand2;
        diff = operand1 - operand2;
        System.out.println(operand1 + " + " + operand2 + " = " + sum);
        System.out.println(operand1 + " - " + operand2 + " = " + diff); }
}
```

Here's the result of this code:

```
C:\>java App
5 + 4 = 9
5 - 4 = 1
```

Shift Operators (>>, >>>, and <<)

You use the shift operators to shift all the bits of a number left or right a specified number of binary places. There are three shift operators—right shift (>>) operator, unsigned right shift (>>>), and left shift (<<) operator. Here's how you use these operators:

```
new_value = value << number_places;
new_value = value >> number_places;
new_value = value >>> number_places;
```

For example, `16 >> 2` shifts the number 16 right by two binary places, which is the same as dividing it by 4; therefore, `16 >> 2` equals 4. You commonly use the shift operators when packing binary values into an `int` or `long` as fields, because you can add a number to the `int` or `long` and then shift it left to make room for the next field of data.

Here's something you should know: The **>>** operator respects the sign of its operand; and because a negative value means that the leftmost bit is 1, shifting a negative number right introduces a new 1 at the left. Therefore, shifting 1111111111111100, which is -4 as a `short`, turns it into 1111111111111110, which is -2. Also, shifting -1, which is 1111111111111111, gives you 1111111111111111, which is still -1. If you really want to work with the actual bits in a number when you shift them right and not have a one added to the left when shifting negative numbers, use the unsigned right shift operator (>>>). This introduces a zero at the left, whether or not the number you are shifting is positive or negative.

Here's an example that puts the shift operators to work:

```
public class App {
        public static void main(String[] args) {
        int value =16, negValue = -1;
        System.out.println(value + " << 2 = " + (value << 2));
        System.out.println(value + " >> 2 = " + (value >> 2));
        System.out.println(negValue + " >> 2 = " + (negValue >> 2));
        System.out.println(negValue + " >>> 22 = " + (negValue >>> 22)); }
}
```

Here's the result of this code:

```
C:\>java App
16 << 2 = 64
16 >> 2 = 4
-1 >> 2 = -1
-1 >>> 22 = 1023
```

Relational Operators (>, >=, <, <=, ==, and !=)

The Big Boss appears and says, "The budget is just about spent, and we need to make sure it doesn't go negative". "Hmm", you say, "sounds like a job for the less than relational operator. Now, about my raise...". "Forget it", says the BB.

Relational operators allow you generate logical conditions that can be used with conditional as well as looping statements. For example, here's how you could check to make sure the budget is greater than zero using a Java `if` statement:

```
public class App {
  public static void main(String[] args) { int budget = 1;
        if (budget < 0) { System.out.println("Uh oh."); }
        else { System.out.println("Still solvent."); }
  }
}
```

Here's the result of this code:

```
C:\>java App
Still solvent.
```

Here's a list of all the relational operators; these operators will return true if their operands match the given descriptions:

- `>` (greater than; for example, `operand1 > operand2` returns true if `operand1` is greater than `operand2`.)
- >= (greater than or equal to)
- < (less than)
- <= (less than or equal to)
- == (equal to)
- != (not equal to)

You can combine the logical conditions you create with a relational operator with the logical operators (see the next topic for the details).

TIP

Here's a Java pitfall to avoid: When you are creating a logical condition, bear in mind that you probably want to use == instead of =. For example, the expression budget == 0 is true if the value in budget is 0, but the expression budget = 0 assigns a value of 0 to budget. Be careful, because using = instead of == in logical conditions is a very common mistake.

Bitwise and Bitwise Logical AND, XOR, and OR (&, ^, and |)

"Help!" the Novice Programmer says. "I need to find out whether bit number 3 of an integer is set to 1. Is there an easy way to do this?" "Sure", you say. "You can use a bitwise operator".

The bitwise operators let you examine the individual bits of values. For example, when you use the `&` bitwise operator with two operands, each bit in one operand is logically *anded* with the corresponding bit in the other operand. If both bits are 1, a one appears in that place in the result; otherwise, a zero will appear in that place. For example, the Novice Programmer could test whether the third bit of a value is set to 1 by *anding* the value with a number for which he knows only one bit—the third bit—is set to 1. If the result of the `AND` operation is not zero, the third bit of the original value was set. Here's how this would look in code:

```
public class App {
  public static void main(String[] args) {
        int value = 12;
        int bit3setting = value & 1 << 3;
        if (bit3setting != 0) { System.out.println("Bit 3 is set."); }
        else { System.out.println("Bit 3 is not set."); }
  }
}
```

Here's the result:

```
C:\>java App
Bit 3 is set.
```

You can find the bitwise operators in Table 3.2. In overview, here's how they work: The OR operator (|) returns 0 when both bits are 0 and returns 1 otherwise. The AND operator (&) returns 1 when both bits are 1 and returns 0 otherwise. Finally, the XOR operator (^, called the *exclusive OR*) returns 1 when one bit is 0 and the other is 1, and it returns 0 otherwise.

When the &, ^, and | operators operate on Boolean (true/false) values, they are considered bitwise *logical* operators. The bitwise logical operators work the same as the bitwise operators (substitute false for 0 and true for 1), as you can see in Table 3.3.

In overview, here's how the bitwise logical operators work: The OR operator (|) returns false when both operands are false, and it returns true otherwise. The AND operator (&) returns true when both operands are true, and it returns false otherwise. The XOR operator (^) returns true when one operand is false and one is true, and it returns false otherwise.

Table 3.2: The OR, AND, and XOR bitwise operators

x	y	x | y (OR)	x & y (AND)	x ^ y (XOR)
0	0	0	0	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	0

Table 3.3: The OR, AND, and XOR bitwise logical operators

X	y	x | y (OR)	x & y (AND)	x ^ y (XOR)
False	False	False	False	False
True	False	True	False	True
False	True	True	False	True
True	True	True	True	False

Here's an example in which we tie two logical conditions together, displaying a message if either is true, using the | operator:

```
public class App {
  public static void main(String[] args) {
        int budget = 1; boolean fired = false;
        if (budget < 0 | fired == true) { System.out.println("Uh oh."); }
        else {  System.out.println("Still solvent."); }
  }
}
```

Here's the result:

```
C:\>java App
Still solvent.
```

In this next example, we insist that the temperature be between 60 and 90 degrees, using the & bitwise logical operator, before printing out a message:

```
public class App {
  public static void main(String[] args) {
  int temperature = 70;
  if (temperature < 90 & temperature > 60) {
        System.out.println("Time for a picnic."); }
  }
}
```

Here's the result:

```
C:\>java App
Time for a picnic.
```

As you can see, the bitwise logical operators can be very useful. Java also includes two logical operators: && and ||. We'll take a look at them next.

Logical (&& and ||)

The two logical operators you usually use in logical expressions are the logical AND (&&) operator and logical OR (||) operator. Table 3.4 shows how these operators work:

Table 3.4: The OR and AND logical operators

x	y	x || y (OR)	x && y (AND)
False	False	False	False
True	False	True	False
False	True	True	False
True	true	true	true

The OR operator (||) returns true if any of its operands is true and false if all operands are false. The AND (&&) operator returns false if any of its operands is false and true only when all operands result in true.

The && and || operators also have another interesting property—they are *short-circuit operators*, which means that if they can determine all they need to know by evaluating the left operand, they won't evaluate the right operand. This is very useful in cases such as the following, where we are testing both whether a value holds 0 and whether its reciprocal is less than 1000. If the value is indeed 0, the second part of the expression, where its reciprocal is calculated, is not executed. This way a divide-by-zero overflow error doesn't occur. Here's the code:

```
public class App {
  public static void main(String[] args) { double value = 0;
        if (value != 0 && 1 / value < 1000) {
                System.out.println("The value is not too small."); }
        else {
                System.out.println("The value is too small."); }
  }
}
```

Here's the result:

```
C:\>java App
The value is too small.
```

The logical operators differ from the bitwise logical operators because the logical operators are short-circuit operators. To see this at work, take a look at the following code, where the assignment in the if statement is performed when we use the & operator but not when we use the && short-circuit operator:

```
public class App {
        public static void main(String[] args) {
  double int1 = 0, int2 = 1, int3 = 1;
  if (int1 != 0 & (int2 = 2) == 1) { }
        System.out.println("int2 = " + int2);
  if (int1 != 0 && (int3 = 2) == 1) { }
        System.out.println("int3 = " + int3);  }
}
```

Here's the result:

```
C:\>java App
int2 = 2.0
int3 = 1.0
```

The if-then-else Operator (?:)

"OK", says the Novice Programmer. "I have mastered the operators. I'm ready for the Java conditional statements". "Not so fast", you say. "What about the ternary conditional operator?" "The *what*?" the NP asks.

There's a Java operator that acts much like an if-else statement—the ternary operator (?:). This operator is called a *ternary operator* because it takes three operands—a condition and two values:

```
value = condition ? value1: value2;
```

If *condition* is true, the ?: operator returns *value1*, and it returns *value2* otherwise.

In this way, the preceding statement works like the following `if` statement:

```
if (condition) { value = value1; }
else { value = value2; }
```

Here's an example where we convert an integer between 0 and 15 into a hexadecimal digit using the `?:` operator. This operator is perfect here, because we can use it to return a string made from the value itself if the value is less than 10 or a letter digit if the value is 10 or greater, like this:

```
public class App {
        public static void main(String[] args) {
        int value = 15;
        String digit, chars[] = {"a", "b", "c", "d", "e", "f"};
        digit = value < 10 ? String.valueOf(value) : chars[value - 10];
        System.out.println(value + " = 0x" + digit); }
}
```

Here's the result:

```
C:\>java App
15 = 0xf
```

Assignment Operators (= and [operator]=)

The most basic operators are the assignment operators, and we have been using these operators throughout the book already. You use the = operator to assign a variable a literal value or the value in another variable as follows:

```
public class App {
        public static void main(String[] args) {
        int value = 12;
        System.out.println("The value = " + value); }
}
```

Here's the result:

```
C:\>java App
The value = 12
```

As in C++, you can perform multiple assignments in the same statement (this works because the assignment operator itself returns the assigned value):

```
public class App {
        public static void main(String[] args) {
        int value1, value2, value3;
        value1 = value2 = value3 = 12;
        System.out.println("value1 = " + value1);
        System.out.println("value2 = " + value2);
        System.out.println("value3 = " + value3); }
}
```

Here's the result:

```
C:\>java App
value1 = 12
value2 = 12
value3 = 12
```

Also, as in C++, you can combine many operators with the assignment operator (=). For example, += is the addition assignment operator, which means *`value += 2`* is a shortcut for *`value = value + 2`*. Here's an example that puts the multiplication assignment operator to work:

```
public class App {
        public static void main(String[] args) {
        int value = 10;
        value *= 2;
        System.out.println("value * 2 = " + value); }
}
```

Here's the result:

```
C:\>java App
value * 2 = 20
```

There are quite a few combination assignment operators. Here's a list of them:

- `%=` (modulus assignment)
- `&=` (bitwise `And` assignment)
- `*=` (multiplication assignment)
- `/=` (division assignment)
- `^=` (bitwise `Xor` assignment)
- `|=` (bitwise `Or` assignment)
- `+=` (addition assignment)
- `<<=` (shift left assignment)
- `<=` (less than or equal to)
- `-=` (subtraction assignment)
- `>>=` (shift right assignment)
- `>>>=` (shift right zero fill assignment)

That completes the list of Java operators, but there's one more popular way of handling math in Java—the `Math` class. This class is part of the `java.lang` package (which the Java compiler imports by default). We'll take a look at this class in the next topic.

Using the **Math** Class

"Hey", the Novice Programmer says, "I want to raise 3 to the power 4, but there's no Java operator for exponentiation". "You can use the `Math` class's `pow()` method", you say. "And by the way, 3 to the power 4 are 81". "I'll believe it when Java tells me so", says the NP.

You can use the `java.lang.Math` class to perform any math operations. For example, here's how to solve the Novice Programmer's problem by using the `Math.pow()` method:

```
public class App {
        public static void main(String[] args) {
   System.out.println("3 x 3 x 3 x 3 = " + Math.pow(3, 4)); }
}
```

The result is:

```
C:\>java App
3 x 3 x 3 x 3 = 81.0
```

The constants and some commonly used methods of the `java.lang.Math` class are as follows:

- `double E`—The constant *e* (2.7182818284590452354)
- `double PI`—The constant *pi* (3.14159265358979323846)
- double sin(double a)—Trigonometric sine
- `double cos(double a)`—Trigonometric cosine
- `double tan(double a)`—Trigonometric tangent
- `double asin(double a)`—Trigonometric arcsine
- `double acos(double a)`—Trigonometric arccosine
- `double atan(double a)`—Trigonometric arctangent
- `double atan2(double a, double b)`—Trigonometric arctangent, two-operand version
- double exp(double a)—Raise *e* to a power
- double log(double a)—Log of a value
- `double sqrt(double a)`—Square root of a value
- double pow(double a, double b)—Raise to a power
- double IEEEremainder(double f1, double f2)—IEEE remainder method

- double ceil(double a)—Ceiling method
- double floor(double a)—Floor method
- double rint(double a)—Random integer
- int round(float a)—Rounds a float
- long round(double a)—Rounds a double
- double random()—Random number
- `int abs(int a)`—Absolute value of an `int`
- `long abs(long a)`—Absolute value of a `long`
- float abs(float a)—Absolute value of a float
- double abs(double a)—Absolute value of a double
- int min(int a, int b)—Minimum of two int types
- long min(long a, long b)—Minimum of two long types
- float min(float a, float b)—Minimum of two float types
- double min(double a, double b)—Minimum of two double types
- int max(int a, int b)—Maximum of two int types
- long max(long a, long b)—Maximum of two long types
- float max(float a, float b)—Maximum of two float types
- double max(double a, double b)—Maximum of two double types

Changes in the **Math** Class

Various new methods are added to `java.lang.Math` class. They are as follows:

- double copySign (double magnitude, double sign)—Copy sign of second argument to first argument
- `double copySign(float magnitude, float sign)`—Copy sign of second argument to first argument
- `int getExponent(double d)`—It returns unbiased exponent used in representation
- `int getExponent(float f)`— It returns unbiased exponent used in representation
- `double nextAfter(double start, double direction)`—It returns floating-point value adjacent to first argument in the direction of second argument
- `float nextAfter(float start, double direction)`—It returns floating-point value adjacent to first argument in the direction of second argument
- `double nextUp(double d)`—Floating-point value adjacent to d in direction of positive infinity
- `float nextUp(float f)`—Floating-point value adjacent to f in direction of positive infinity
- double scalb(double d, int scaleFactor)—It returns d x 2scaleFactor
- float scalb(float f, int scaleFactor)—It returns f x 2scaleFactor

Class **StrictMath**

The StrictMath class consists of methods that let you perform basic numeric operations, such as elementary exponential, logarithm, square root, and trigonometric functions. The java.lang.StrictMath class is defined with respect to fdlibm version 5.3. For this class, use IEEE 754 core function version (residing in a file whose name begins with letter e) where fdlibm provides more than one definition for a function (such as acos). The methods that require fdlibm semantics are sin, cos, tan, asin, acos, atan, exp, log, log10, cbrt, atan2, pow, sinh, cosh, tanh, hypot, expm1, and log1p. This ensures the portability of Java program as these methods produce same result across different platforms. The description of the methods of the StrictMath class is as follows:

- `static double cbrt (double a)`—It returns the cube root of a double value.

- `static double cosh(double x)`—It returns the hyperbolic cosine of a double value.
- static double expm1(double x)— It returns ex -1.
- `static double hypot(double x, double y)`—It returns sqrt(x2 +y2) without intermediate overflow or underflow.
- static double log(double a)—It returns the natural logarithm (base e) of a double value.
- static double log10(double a)—It returns the base 10 logarithm of a double value.
- static double log1p(double x)—It returns the natural logarithm of the sum of the argument and 1.
- `static double rint(double a)`—It returns the double value that is closest in value to the argument and is equal to a mathematical integer
- `static double signum(double d)`—It returns the signum function of the argument; zero if the argument is zero; 1.0 if the argument is greater than zero; -1.0 if the argument is less than zero.
- static float signum(float f)— It returns the signum function of the argument; zero if the argument is zero, 1.0f if the argument is greater than zero, -1.0f if the argument is less than zero.
- `static double sinh(double x)`—It returns the hyperbolic sine of a double value.
- `static double tanh(double x)`—It returns the hyperbolic tangent of a double value
- `static double toDegrees(double angrad)`—It converts an angle measured in radians to an approximately equivalent angle measured in degrees
- `static double toRadians(double angdeg)`—It converts an angle measured in degrees to an approximately equivalent angle measured in radians.
- `static double ulp(double d)`—It returns the size of an ulp of the argument.
- static float ulp(float f)—It returns the size of an ulp of the argument

Comparing Strings

When you are working with the `String` class, there are some methods you can use much like operators. For example, you can use the `equals` method, `equalsIgnoreCase()` method, and `compareTo()` method as follows:

- `s1.equals(s2)`—It returns true if `s1` equals `s2`.
- `s1.equalsIgnoreCase(s2)`—It returns true if `s1` equals `s2` (ignoring case).
- `s1.compareTo(s2)`—It returns a value less than zero if `s1` is less than `s2` lexically, returns zero if `s1` equals `s2`, or returns a value greater than zero if `s1` is greater than `s2`.

Here's an example putting these methods to work:

```
public class App {
        public static void main(String[] args) {
        String s1 = "abc";
        String s2 = "abc";
        String s3 = "ABC";
        String s4 = "bcd";
        if (s1.equals(s2)) { System.out.println("s1 == s2"); }
        else { System.out.println("s1 != s2"); }
        if (s1.equalsIgnoreCase(s3)) {
        System.out.println("s1 == s3 when ignoring case"); }
        else { System.out.println("s1 != s3 when ignoring case"); }
        if (s1.compareTo(s4) < 0) { System.out.println("s1 < s2"); }
        else if (s1.compareTo(s4) == 0){ System.out.println("s1 == s2"); }
        else if (s1.compareTo(s4) > 0 ) { System.out.println("s1 > s2"); }
  }
}
```

Here's the result of this code:

```
C:\>java App
s1 == s2
s1 == s3 when ignoring case
s1 < s2
```

The **if** Statement

"Hmm", says the Novice Programmer. "I'm stuck. I want to write an absolute value routine in Java, and I don't know how to proceed". "I don't suppose you've ever heard of the `Math` class's `abs` method?" you reply. "The *what*?" asks the NP.

The `if` statement lets you execute a statement or a set of statements depending upon the result of test condition. However, if the test condition evaluates false, no statement is executed. In general, the `if` statement is used as follows:

```
if (condition)
  statement;
```

Note that *`statement`* can be a compound statement also, which means it can be made up of a number of statements enclosed in curly braces. Here's an answer to what the Novice Programmer was trying to do. Start by checking whether the value is greater than zero; and if so, just print out the value itself, as done in the following code:

```
public class App {
        public static void main(String[] args) {
        int value = 10;
        if(value > 0)
        System.out.println("Abs(" + value + ") = " + value); }
}
```

Here's the result of this code:

```
C:\>java App
Abs(10) = 10
```

Another example for the Novice Programmer where an `if` statement is used to execute multiple statements is as follows:

```
public class App {
  public static void main(String[] args) {
        int value = 10;
        if(value > 0) {
        System.out.println("The number was positive.");
        System.out.println("Abs(" + value + ") = " + value); }
  }
}
```

Here's the result of this code:

```
C:\>java App
The number was positive.
Abs(10) = 10
```

The **else** Statement

The `else` statement lets you specify statement(s) which gets executed when the `if` condition evaluates to false. Here's an example using the `else` statement, which can also display absolute value of negative numbers:

```
public class App {
        public static void main(String[] args) {
        int value = -10;
        if(value > 0) {
        System.out.println("Abs(" + value + ") = " + value); }
        else {
        System.out.println("Abs(" + value + ") = " + -value); }
  }
}
```

Here's the result of this code:

```
C:\>java App
Abs(-10) = 10
```

Nested **if** Statement

The nested if statement allows that one `if` statement can be written inside another `if` statement. Here's an example to show how a nested `if` statement works:

```
public class App {
        public static void main(String[] args) {
        double value = 2;
        if (value != 0) {
        if (value > 0)
                System.out.println("The result = " + (1 / value));
        else
                System.out.println("The result = " + (-1 / value));     }
  }
}
```

Here's the result of this code:

```
C:\>java App
The result = 0.5
```

The **if-else** Ladders

An entire sequence of the `if-else` statements is called an `if-else ladder`. Here's an example to show how an `if-else` ladder works:

```
public class App {
  public static void main(String[] args) {
        String month = "March";
        if(month == "January")
                System.out.println("It\'s January");
        else if (month == "February")
                System.out.println("It\'s February");
        else if (month == "March")
                System.out.println("It\'s March");
        else if (month == "April")
                System.out.println("It\'s April");
        else if (month == "May")
                System.out.println("It\'s May"); }
}
```

In this case, we test the value in a string variable successively until we find a match to the current year of the calendar (we are testing only for the first five months). Here's the result of this code:

```
C:\>java App
It's March
```

Note that although it's possible to create `if-else` ladders in this way, Java actually includes a statement expressly for situations like this—the `switch` statement. We'll take a look at this statement in the next topic.

The **switch** Statement

"Good Lord?", says the Novice Programmer. "I'm getting tired of writing `if-else` ladders—the one in my program must be five pages long now". "How about trying a `switch` statement?" you ask. "What's that?" the NP asks.

The `switch` statement is Java's multipath branch statement; it provides the same kind of functionality as an `if-else` ladder does (see the previous topic) but in a form that's much easier to work with. Here's what the `switch` statement looks like in general:

```
switch (expression) {
        case value1:
        statement1;
        [break;]
```

```
        case value2:
   statement2;
        [break;]
        case value3:
   statement3;
        [break;]
. . .
        default:
default_statement;
}
```

Here, the value of *expression*, which must be of type `byte`, `char`, `short`, or `int`, is compared against the various test values in the `case` statements: *value1*, *value2*, and so on. If the expression matches one of the `case` statements, the code associated with that `case` statement—*statement1*, *statement2*, and so on—is executed. If execution reaches a `break` statement, the `switch` statement ends. Here's an example in which we display the day of the week based on a numeric value by using a `switch` statement:

```
public class App {
   public static void main(String[] args) {
        int day = 5;
        switch(day) {
                case 0:
                        System.out.println("Today is Sunday."); break;
                case 1:
                        System.out.println("Today is Monday."); break;
                case 2:
                        System.out.println("Today is Tuesday."); break;
                case 3:
                        System.out.println("Today is Wednesday."); break;
                case 4:
                        System.out.println("Today is Thursday."); break;
                case 5:
                        System.out.println("Today is Friday."); break;
                default:
                        System.out.println("Today is Saturday.");       }
   }
}
```

Here's the result of this code:

```
C:\>java App
Today is Friday.
```

Try to execute the preceding example without writing the `break` statements and see what it returns as the day today. You'll find that it returns `Today is Friday.` and `Today is Saturday`. This signifies that if you don't specify a `break` statement at the end of a `case` statement, the execution will continue with the code in the next `case` statement. This may prove useful when you want to execute the same code for multiple case test values, as shown in the following code:

```
public class App {
   public static void main(String[] args) {
        int temperature = 78;
        switch(temperature) {
                case 60:
                case 61:
                case 62:
                case 63:
                case 64:
                        System.out.println("Very cold"); break;
                case 65:
                case 66:
                case 67:
                case 68:
                case 69:
                        System.out.println("It\'s cool"); break;
                case 70:
                case 71:
```

```
                    case 72:
                    case 73:
                    case 74:
                    case 75:
                            System.out.println("It\'s warm"); break;
                    default:
                            System.out.println("Uff, very hot!! "); } }
    }
```

Here's the result of this code:

```
    C:\>java App
    Uff, very hot!!
```

Note that a default case executes if none of the above cases match or when no break statements are specified.

Using Strings in switch Statement

The `switch` statement has been upgraded in Java 7. Now, you can also use Strings inside the `switch` statements. The following is an example to make it clearer to you:

```
    public class App {
      public static void main(String[] args) {
            String country="India";
            switch(country) {
                    case "India": System.out.println("India");break;
                    case "China": System.out.println("China");break;
                    default: System.out.println("Neither India nor China");} }
    }
```

Here's the result of this code:

```
    C:\>java App
    India
```

where the first case matches with the `switch` statement and therefore executed, and `India` gets printed.

The **while** Loop

"Well", says the Novice Programmer, "I'm in trouble again. The Big Boss wants me to create a commercial program that will calculate factorials, and I don't even know what a factorial is!" "Well", you say, "six factorial, written as '6!', is equal to 6 × 5 × 4 × 3 × 2 × 1. And you can write your program with a `while` loop".

A `while` loop executes a statement(s) inside a loop body as long as the loop test condition evaluates to true. In general, a `while` loop looks as follows:

```
    while(condition)
     statement;
```

Note that the loop is executed not even once if the test condition evaluates to false. An example that shows how a `while` loop works is as follows:

```
    public class App {
            public static void main(String[] args) { int value = 10;
            while (value > 0) {
            System.out.println("Current value = " + value--); }     }
    }
```

In this case, we display a value, successively subtract 1 from that value and then display the result. When the value becomes 0, the `while` loop stops because the condition (`value > 0`) has become false. In other words, the loop executes till `value` remains positive.

Here's what this `while` loop returns:

```
    C:\>java App
    Current value = 10
    Current value = 9
    Current value = 8
    Current value = 7
    Current value = 6
    Current value = 5
    Current value = 4
```

```
Current value = 3
Current value = 2
Current value = 1
```

Here's an example where a `while` loop has been used to calculate the factorial of 6:

```
public class App {
  public static void main(String[] args) {
        int value = 6, factorial = 1, temp;
        temp = value; //make a destructive copy.
        while (temp > 0) { factorial *= temp--; }
        System.out.println(value + "! = " + factorial); }
}
```

Here's how the program calculates the factorial of 6:

```
C:\>java App
6! = 720
```

A more advanced example is as follows:

```
import java.util.*;
public class App { public static void main(String[] args) {
        int value = 32, temp = value;
        StringBuffer sb = new StringBuffer();
        Stack <String> st = new Stack <String>();
        while (temp > 0) {
        st.push(String.valueOf(temp % 16));
        temp >>>= 4; }
        while(!st.empty()) { sb.append(new String((String) st.pop())); }
System.out.println("Converting " + value + " to Hexadecimal yields 0x" + sb); }
}
```

In this case, we are converting a number into hexadecimal by successively stripping off hex digits with the modulus operator. Because the digits come out in reverse order, we are using a `while` loop to push them onto a Java *stack*, which you'll see when we discuss the collection classes. After pushing the digits onto the stack, we pop them in another `while` loop to reverse the order of the digits and create the `StringBuffer` object to display.

Here's what this program's output looks like:

```
C:\>java App
Converting 32 to Hexadecimal yields 0x20
```

Here's a fact that can come in handy: Because `null` statements are valid in Java, a `while` loop doesn't have to have a body at all. Here's an example showing a crude way of calculating an integer square root (note that all the work here takes place in the condition part of the loop):

```
public class App { public static void main(String[] args) {
        int target = 144, sqrt = 1;
        while (++(sqrt) * (sqrt) != target) ;
        System.out.println("sqrt(" + target + ") = " + sqrt); }
}
```

Here's the result:

```
C:\>java App
sqrt(144) = 12
```

Another type of `while` loop—the `do-while` loop—is discussed in the next topic.

The **do-while** Loop

The Programming Correctness Czar says, "So, you have a `while` loop in Java. In C++, we have both a `while` loop and a `do-while` loop". "That's funny", you say. "We have both of those in Java, too".

The `do-while` loop is similar to a `while` loop. It is also used to repeat a set of statements a number of times. However, the difference lies where the test condition is evaluated. A `do-while` loop tests the condition after the loop body is executed, whereas a `while` loop tests the condition before the loop body executes. Both loops terminate once the test condition evaluates to false. A `do-while` loop looks as follows:

```
do
   statement;
while(condition);
```

It can be seen in the preceding lines that a `do-while` loop will always execute at least once even if the condition evaluates to false. This is because the condition is tested after the loop body gets executed. Here's an example to show how a `do-while` loop works:

```
public class App {
        public static void main(String[] args) {
        int values[] = {1, 2, 3, 0, 5}, test, index = 0;
        do {
        test = 5 * values[index++];
        System.out.println(test);
   } while (test < 15); }
}
```

The result is as follows:

```
C:\>java App
5
10
15
```

Question arises when to use a `while` loop and when to use a `do-while` loop? The answer is whenever the situation demands that looping statements should execute `only` if the condition is true, you use a `while` loop. The following code makes it clearer, where a `do-while` loop evaluates the reciprocal of a value but can only test whether the value is a nonzero value at the end of the loop:

```
public class App {
        public static void main(String[] args) {
        double value = 0;
        do {
        System.out.println("The reciprocal = " + 1 / value);
        } while (value > 0);
        }
}
```

It's far better in this case to use a `while` loop to test for 0 first:

```
public class App {
        public static void main(String[] args) {
        double value = 0;
        while (value > 0) {
        System.out.println("The reciprocal = " + 1 / value); }
        }
}
```

The **for** Loop

The Novice Programmer is back and says, "I like `while` loops, but they are not the easiest to work with when handling arrays—I really need a numeric index there. Is there anything else I can use?" "Certainly", you say, "try a `for` loop".

The Java `for` loop is a good choice when you want to use a numeric index that you automatically increment or decrement each time through the loop, such as when you are working with an array. Here's what the `for` loop looks like in general (note that *`statement`* can be a compound statement, including several single statements inside curly braces):

```
for (initialization_expression; test_conditon; iteration_expression) {
        statement
}
```

`Initialization_expression` lets you initialize the loop index, `test_condition` signifies a conditional expression, such as `loop_index<=5`, and `iteration_expression` increments or decrements the loop index. You can also use multiple loop indexes in a `for` loop.

An example to show how a `for` loop works is as follows:

```
public class App {
        public static void main(String[] args) {
        int loop_index;
        for (loop_index = 1; loop_index <= 5; loop_index++) {
```

```
            System.out.println("This is iteration number " + loop_index); }
            }
}
```

The loop index is initialized with value 1, the loop terminates when the index exceeds 5, which means the loop is executed only 5 times, and every time the loop index is incremented by 1:

Here's the result:

```
C:\>java App
This is iteration number 1
This is iteration number 2
This is iteration number 3
This is iteration number 4
This is iteration number 5
```

Here's an example that finds the average student score by looping over all the scores and summing them (note that we are actually declaring and initializing the loop index to 0 in the initialization expression):

```
public class App {
        public static void main(String[] args) {
        double grades[] = {88, 99, 73, 56, 87, 64};
        double sum, average;
        sum = 0;
        for (int loop_index = 0; loop_index < grades.length;
        loop_index++) {
        sum += grades[loop_index]; }
        average = sum / grades.length;
        System.out.println("Average grade = " + average); }
}
```

Here's the result of this code:

```
C:\>java App
Average grade = 77.83333333333333
```

When you declare a loop variable (such as `loop_index` in this example), the *scope* of that variable is limited to the `for` loop body (the scope of a variable is the part of the program that you can access it in, as you'll see in the next chapter).

Note that you can use very general expressions in a `for` loop. Java lets you separate expressions in a `for` loop with a comma, as shown in the following example in which we are using two loop indexes:

```
public class App {
        public static void main(String[] args) {
        for (int loop_index = 0, doubled = 0; loop_index <= 10;
        loop_index++, doubled = 2 * loop_index) {
        System.out.println("Loop index " + loop_index +" doubled equals " + doubled); }
  }
}
```

Here's the result of this code:

```
C:\>java App
Loop index 0 doubled equals 0
Loop index 1 doubled equals 2
Loop index 2 doubled equals 4
Loop index 3 doubled equals 6
Loop index 4 doubled equals 8
Loop index 5 doubled equals 10
Loop index 6 doubled equals 12
Loop index 7 doubled equals 14
Loop index 8 doubled equals 16
Loop index 9 doubled equals 18
Loop index 10 doubled equals 20
```

You don't have to give a `for` loop to anybody at all—in fact, you can use a `null` statement. Here's an example in which we are summing all the elements of an array in a `for` loop without any code in its body:

```
public class App {
        public static void main(String[] args) {
        int array[] = {1, 2, 3, 4, 5}, sum = 0;
        for (int loop_index = 0; loop_index < array.length;
        sum += array[loop_index++]);
```

```
            System.out.println("The sum = " + sum); }
    }
```

Here's the result of this code:

```
    C:\>java App
    The sum = 15
```

You can even turn a `for` loop into a `while` loop. Here's an example that's adapted from the previous factorial example in the topic ("The `while` Loop"):

```
    public class App {
            public static void main(String[] args) {
            int value = 6, factorial = 1, temp;
            temp = value;
            for( ;temp > 0; ) { factorial *= temp--; }
            System.out.println(value + "! = " + factorial); }
    }
```

Here's the result of this code:

```
    C:\>java App
    6! = 720
```

The **for-each** Loop

The NP says, "`For` loops are good; but if I have to deal with a condition as well, it becomes a pain. Is this all that is available in Java?". "No", you say. "You have the `for-each` loop for this situation".

Further to the `for` loop, a new construct called the `for-each` loop is available. This enhanced construct allows you to automatically play through the elements of a collection or an array. The `for-loop` construct does the looping automatically, which would otherwise have to be done manually earlier. In its place, you can simply specify the array to iterate through and a variable to access each element, as shown here:

```
    for (variable: collection)
          statement;
```

To display the output, the program shown here prints out each element passed in along the command line:

```
    public class Args {
            public static void main(String args[]) {
            for (String element: args) { System.out.println(element); } }
    }
```

Compile it with your compiler and then pass in some random set of command-line arguments:

```
    C:\javac Args.java
    C:\java Args Kogent Learning Solutions
```

You'll get the provided arguments printed out to standard output, one per line:

```
    Kogent
    Learning
    Solutions
```

Although this is nothing great, it avoids the manual writing of the code where you had to update the index variable yourself:

```
    for (int k=0; k < args.length; k++) { System.out.println(args[k]); }
```

You can also perform the preceding program by initializing the values in the program itself instead of giving values at runtime. Let's write another program using enumeration. This is created by using the new `enum` keyword:

```
    public class App { enum Country {India,Japan,China}
      public static void main (String[] args) {
            for(Country country : Country.values()) {
                    System.out.println("The country is " +country); } }
    }
```

The identifiers with the `enum` statement, i.e. India, Japan and China, are called enumeration constants. Their type is the type of the enumeration in which they are declared, i.e., `Country`. Once you have created an enumeration, you can create a variable of the type:

```
    for(Country country : Country.values())
```

Here, `country` is a variable of enumeration type `Country`. Each of the values in the enumeration is passed to the variable and thus display on the screen:

```
C:\> java App
The country is India
The country is Japan
The country is China
```

The construct "`for (variable: collection)`" doesn't add any new keywords to the language. Alternatively, it extends the meaning of the basic `for` loop. This `for-each` loop says, for each element in the "collection", assigns it to the variable "element", and executes the statement.

The construct works for both arrays and implementations of the `Collection` interface, like `ArrayList`, `HashSet`, and the keys from a Map like a `Properties` object. As you can infer from their names, the collection classes let you group elements in various ways. The collection classes also define various methods that make working with those items easier. Collection was finally added by J2SE 1.2. With the new versions of JDK, the power of Collection Framework has significantly increased and its use has been streamlined.

In fact, it has simplified the way you track a group of objects. A few simple methods are needed to use collection classes. You'll learn more about the collections in Chapter 21. Before moving onto the program, let's learn about the most basic collection class, `ArrayList`. An `ArrayList` is similar to an array but avoids many of the most common problems of arrays. For example, an array can't grow or shrink in size once created; therefore, you must know in advance the size of an array, whereas an ArrayList supports dynamic arrays that can grow or shrink in size as per the need.

The following program shows the case of using the `for-each` construct with a `Collection`:

```
import java.util.ArrayList;
public class ForLoopTest {
        public static void main(String[] args)
        //args the command line  arguments
        {
        double[] array =  {2.5, 5.2, 7.9, 4.3, 2.0, 4.1, 7.3, 0.1, 2.6};
        //leave details of the loop such as indices out of the picture
        for(double d: array) {   System.out.println(d); }
        System.out.println("---------------------");
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(7);
        list.add(15);
        list.add(-67);
        for(Integer number: list) { System.out.println(number); }
        System.out.println("---------------------");
        //Works identically with autounboxing
        for(int item: list) { System.out.println(item); }
                System.out.println("---------------------");    }
}
```

Look at the line here as you can specify the type of elements the array list is allowed to contain. This statement creates an array list that holds `Integer` objects:

```
ArrayList<Integer> list = new ArrayList<Integer>()
```

Here's the output of this code:

```
C:\>java ForLoopTest
2.5
5.2
7.9
4.3
2.0
4.1
7.3
0.1
2.6
---------------------
7
```

```
15
-67
--------------------
7
15
-67
```

Supporting **for-each** in Your Own Class

"Help!" says the NP "I'm trying to build my own class and handling the array is very tough. Can I have a simpler way of doing the same". "You can use the `for-each` construct here also," you say.

Imagine that you are building a `Catalog` class. A `Catalog` object collects any number of `Product` objects. You store these objects by using an `ArrayList` instance variable defined in `Catalog`. The program will have to iterate with a `Catalog` object frequently through the entire list of products, to do something with each object in turn. Shown here is an example of the `Catalog` class:

```
import java.util.*;
class Catalog{
        private List<Product> products = new ArrayList<Product>();
        void add(Product product) { products.add(product); }
}
```

The `Product` class includes a method that allows you to discount the price on a product:

```
import java.math.BigDecimal;
class Product {
        private String id;
        private String name;
        private BigDecimal cost;
        Product(String id, String name, BigDecimal cost) {
        this.id = id; this.name = name;
        this.cost = cost; }
        void discount(BigDecimal percent) {
        cost = cost.multiply(new BigDecimal("1.0").subtract(percent)); }
  public String toString() { return id + ": " + name + " @ " + cost; }
  }
```

A (Poor) Solution

To allow the client code to work with all products, you could create a method in the `Catalog` that returned the `ArrayList` of products:

```
List<Product> getProducts() { return products; }
```

For choosing an object in the list the client code would have to iterate through the list. However, returning a collection to a client is not feasible. If this is done, the client can modify the list in your collection, as you have given the control over the contents of that collection to the client. And hence, the client code could add or remove elements to the collection without the knowledge of the `Catalog` object. Moreover, the client will have to do more work than necessary.

*Significance of **for-each***

The `for-each` loop provides a simple, consistent solution for iterating arrays, collection classes, and even your own collections. It helps eliminate much of the repetitive code that you would otherwise require. They also eliminate the need for casting as well as other potential problems related to iteration. This is one such addition to Java that was long overdue.

The `for-each` loop can be used with both collections and arrays. The basic function is to simplify the most common form of iteration, where the iterator or index is used solely for iteration, and not for any other kind of operation, such as removing or editing an item in the collection or array. When there is a choice, the `for-each` loop should be preferred over the `for` loop, since it increases legibility. Here is an example on `for-each` loop:

```
import java.util.*;
import java.math.BigDecimal;
public final class For_each_Examples {
```

```
    public static void main(String[] aArgs) {
        List<Number> numbers = new ArrayList<Number>();
        numbers.add(new Integer(42));
        numbers.add(new Integer(-30));
        numbers.add(new BigDecimal("654.2"));
        for ( Number number : numbers ) { log(number); }
        String[] names = {"Ethan Hawke", "Julie Delpy"};
        for( String name : names ) { log("Name : " + name); }
        //iterator :
        Collection<String> words = new ArrayList<String>();
        words.add("Il ne lui faut que deux choses:");
        words.add("le");
        words.add("pain");
        words.add("et");
        words.add("le");
        words.add("temps");
        words.add("- Alfred de Vigny");
        for(Iterator<String> iter = words.iterator(); iter.hasNext();) {
            if (iter.next().length() == 4) { iter.remove(); } }
        log("Edited words: " + words.toString());
        Collection <String> stuff = new ArrayList <String> ();
            stuff.add("blah");
        for (Object thing : stuff) {
            String item = (String)thing;
            log("Thing : " + item); }
    }
    private static void log(Object aThing) { System.out.println(aThing); }
}
```

Here's the result of the code:

```
C:\>java For_each_Examples
42
-30
654.2
Name : Ethan Hawke
Name : Julie Delpy
Edited words : [I1 ne lui faut que deux choses:, le, et, le, temps, - Alfred de Vignyl
Thing : blah
```

Nested Loops

"I'm working with a two-dimensional array", the Novice Programmer says, "and I almost wish I could have a loop *within* a loop, so I could loop over both dimensions". "Of course you can use loops within loops", you reply.

Java lets you nest loops, one within another. Here's an example showing how this works (in this case, we are finding the average value of the elements in a two-dimensional array by looping over all the elements with two `for` loops):

```
public class App {
        public static void main(String[] args) {
    double array[][] = {{1, 2, 3}, {3, 2, 1}, {1, 2, 3}};
    int sum = 0, total = 0;
    for(int outer_index = 0; outer_index < array.length;
    outer_index++) {
        for(int inner_index = 0; inner_index <
    array[outer_index].length; inner_index++) {
        sum += array[outer_index][inner_index];
        total++; } }
    System.out.println("Average array value = " + (sum / total)); }
}
```

Here's the result of this code:

```
C:\>java App
Average array value = 2
```

Using the **break** Statement

The Novice Programmer has another problem: "I have got a multidimensional array that I'm looping over, and sometimes deep inside five-nested loops the results exceed the maximum allowable value, so I want to end all the loops. How the heck do I do that without letting them all finish naturally?" "By using the `break` statement", you reply.

Some languages include a `goto` statement that you can use to jump to any statement you want to in your code, but most languages consider `goto` too unstructured—as does Java, which does not include a `goto` statement. However, a `goto` statement lets you jump out of a loop that's no longer useful, which is a valid thing to do. Because Java doesn't have a `goto` statement you can use to do that, it supports the `break` statement for this purpose.

You can use the `break` statement to end a loop, as in this case where we are ending a loop if a sum becomes greater than 12:

```
public class App {
  public static void main(String[] args) {
        double array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; int sum = 0;
        for(int loop_index = 0; loop_index < array.length; loop_index++) {
                sum += array[loop_index];
                if (sum > 12) break;
                System.out.println("Looping.."); }
                System.out.println("The sum exceeded the maximum value"); }
}
```

Here's the result of this code:

```
C:\>java App
Looping..
Looping..
Looping..
Looping..
The sum exceeded the maximum value
```

What if you have multiple loops that you want to break? In that case, you can label the loops and indicate the one you want to break out of. Here's an example in which we break out of a double-nested loop:

```
public class App {
  public static void main(String[] args) {
        double array[][] = { {1, 2, 3}, {3, 2, 1}, {1, 2, 3}}; int sum = 0;
outer: for(int outer_index = 0; outer_index < array.length; outer_index++) {
    for(int inner_index = 0; inner_index <
  array[outer_index].length; inner_index++) {
                        sum += array[outer_index][inner_index];
                        if (sum > 3) break outer; }
                System.out.println("I'm not going to print."); }
        System.out.println("The loop has finished.");   }
}
```

Here's the result of this code:

```
C:\>java App
The loop has finished.
```

Note that if you don't use a label with the `break` statement, you'll only break out of the current loop.

Using the **continue** Statement

"I like loops", the Novice Programmer says. "I really do. There's just one problem, though. Sometimes when I'm looping, I come across a value that I don't want to use and I just want to skip to the next iteration of the loop without executing any more code. Can I do that?" "Yes, indeed", you say. "You can use the `continue` statement".

To skip to the next iteration of a loop, you can use the `continue` statement. Here's an example in which we are printing out reciprocals, and we want to avoid trying to print out the reciprocal of 0. If the current loop index equals 0, we skip the current iteration and move on to the next one. Here's the code:

```
public class App {
        public static void main(String[] args) {
  for(double loop_index = 5; loop_index > -5; loop_index--) {
        if (loop_index == 0) continue;
        System.out.println("The reciprocal of " + loop_index +
  " = " + (1 / loop_index)); }
   }
}
```

Here's the result of this code (note that this output skips over the line where the code would try to calculate the reciprocal of 0):

```
C:\>java App
The reciprocal of 5.0 = 0.2
The reciprocal of 4.0 = 0.25
The reciprocal of 3.0 = 0.3333333333333333
The reciprocal of 2.0 = 0.5
The reciprocal of 1.0 = 1.0
The reciprocal of -1.0 = -1.0
The reciprocal of -2.0 = -0.5
The reciprocal of -3.0 = -0.3333333333333333
The reciprocal of -4.0 = -0.25
```

Summary

In this chapter, we have discussed about the operators, conditionals, jump statements, and loops. You also understood how to work with operators, use the `Math` Class, the `StrictMath` Class, compare strings, use strings in switch statement, and use the conditional statements. You have also learned about the implementation of iterative control structure. Towards the end, you have studied about the nested loops and the jump statements.

The next chapter will discuss about object oriented programming.



4

Object-Oriented Programming

If you need an immediate solution to:	*See page:*
Declaring and Defining Classes	111
Declaring and Creating Objects	112
Creating Instance Variables	114
Setting Variable Access	114
Creating Class Variables	115
Creating Methods	116
Setting Method Access	116
Passing Parameters to Methods	117
Command-Line Arguments Passed to main() method	118
Returning Values from Methods	118
Creating Class Methods	119
Creating Data Access Methods	120
Creating Constructors	120
Passing Parameters to Constructors	121
A Full Class Example	121
Understanding Variable Scope	122
Using Recursion	123
Garbage Collection and Memory Management	123
Garbage Collection and the finalize() Method	125
Overloading Methods	125
Overloading Constructors	126
Passing Objects to Methods	126
Passing Arrays to Methods	127
Using the this Keyword	128

If you need an immediate solution to:	*See page:*
Returning Objects from Methods	129
Returning Arrays from Methods	129
The ProcessBuilder and StringBuilder Classes	129
Catching an Exception	132
Nesting the try Statements	136
Using the finally Clause	137
Throwing Exceptions	138
Creating a Custom Exception	139
Catching Multiple Exception Types	139
Rethrowing Exceptions with Improved Type Checking	140
Debugging Java Programs	141

In Depth

This chapter is all about a topic central to any Java program—Object-Oriented Programming (OOP). We first discussed Object-Oriented Programming in Chapter 1 because you cannot write Java code without it. Now that you have come up through the basics of Java syntax, you are ready to work with Object-Oriented Programming in a formal way.

Object-Oriented Programming is really just another technique to let you implement that famous programming dictum—divide and conquer. The idea is that you encapsulate data and methods into objects, making each object semi-autonomous, enclosing private (i.e., purely internal) data and methods in a way that stops them from cluttering the general namespace. The object can then interact with the rest of the program through a well-defined interface as implemented by its public (i.e., externally callable) methods.

Object-Oriented Programming was first created to handle larger programs by breaking them up into functional units. It takes the idea of breaking a program into subroutines one step further, because objects can have both multiple subroutines and data inside them. The result of encapsulating parts of your program into an object is that it is easily conceptualized as a single item, instead of you having to deal with all that makes up that object internally.

Imagine how your kitchen would look filled with pipes, pumps, a compressor, and all kinds of switches used to keep food cold. Every time the temperature of the food got too high, you would have to turn on the compressor and open valves and start cranking the pumps manually. Now wrap all that functionality into an object—a refrigerator—in which all those operations are handled internally, with the appropriate feedback between the parts of the object handled automatically inside the object. That's the idea behind encapsulation—taking a complex system that demands a lot of attention and turning it into an object that handles all its own work internally and can be easily conceptualized, much like a refrigerator. If the first dictum of Object-Oriented Programming is "divide and conquer," the second is surely "out of sight, out of mind."

In Java, Object-Oriented Programming revolves around a few key concepts—classes, objects, data members, methods, and inheritance. Here's what those terms mean in overview:

- **Class**—This is a template from which you can create objects. The definition of a class includes the formal specifications for the class, and any data and methods in it.
- **Object**—This is an instance of a class, much as a variable is an instance of a data type. You can think of a class as the type of an object, and you can think of the object as an instance of a class. Objects encapsulate methods and instance variables.
- **Data members**—These are variables that are part of a class. You use them to store the data the object uses. Objects support both instance variables whose values are specific to the object, and class variables whose values are shared among the objects of that class.
- **Method**—This is a function built into a class or an object. You can have instance methods and class methods. You use instance methods with objects, but you can use a class method just by referring to the class by name—no object is required.
- **Inheritance**—This is the process of deriving one class, called the derived class, from another, called the base class, and being able to make use of the members of the base class in the derived class.

All these constructs are important to Object-Oriented Programming, and we'll get into more details on each of them now.

NOTE

If you are used to working with Object-Oriented Programming in C++, it may surprise you to learn that although Java programs are object-oriented, the object-oriented support in Java is less than what's available in languages, such as C++. For example, the designers of Java decided to let programmers overload methods but not operators (although Java itself overloads operators, such as + for the String class). Also, Java does not support destructors and multiple inheritances directly—instead Java interfaces are used.

Classes

In Object-Oriented Programming, *classes* provide a sort of template for objects. That is, if you think of a class as a cookie cutter, the objects you create from it are the cookies. You can consider a class an object's *type*—you use a class to create an object and then you can call the object's methods from your code.

To create an object, you call a class's *constructor*, which is a method with the same name as the class itself. This constructor creates a new object of the class. We have been creating classes throughout this book already; each time you create a Java program, you need a class. For example, here's how to create a class named `App`, which is stored in a file named `App.java` (this class creates a Java application):

```
public class App { //Object-Oriented Programming
   public static void main(String[] args) {
          System.out.println("Hello from Java!"); }
}
```

When you use the Java compiler, this file, `App.java`, is translated into the bytecode file `App.class`, which holds the complete specification for the `App` class. So, how do you create objects from classes? Take a look at the next section.

Objects

In Java, you call an instance of a class an *object*. To create an object, you call a class's *constructor*, which has the same name as the class itself. Here's an example in which we create an object from the Java `String` class, passing the string we want to enclose in that object to the `String` class's constructor:

```
String s = new String("Hello from Java!");
```

You'll see more about creating objects with constructors throughout this chapter. So, what do you do with an object when you have one? You can interact with it using its data members and methods; take a look at the next two sections.

Data Members

Data members of an object are called *instance data members* or *instance variables*. Data items shared by all objects of a class are called *class data members* or *class variables*. You'll see how to create both instance variables and class variables in this chapter. Data members can be accessible outside an object, or you can make them internal to the object for the private use of the methods inside the object.

Here's an example showing how you might use an object's data member; say you have a class named `Data_class`, and you create an object of this class named `data1`:

```
Data_class data1 = new Data_class("Hello from Java!");
. . .
```

If `Data_class` defines a publicly accessible data member named `data`, you can refer to the data member of `data1` by using the dot operator (`.`), like this:

```
data1.data
```

That means you can print out the data in `data1`, like this:

```
Data_class data1 = new Data_class("Hello from Java!");
System.out.println(data1.data);
```

In this way, you can refer to the data members of an object that the object makes publicly accessible. On the other hand, recall that data hiding is one of the motivations behind Object-Oriented Programming, and giving code outside an object access to the internal data of an object might not be a good idea. Instead, you often give code outside an object access to the object's data only through the object's methods (which mean you can control the object's interface to the rest of the program, checking data values before those values are stored in the object's data members).

Methods

Methods are the functions built into a class and therefore built into the objects you create from that class. You usually divide methods into those intended for use inside the class, called *private methods*, those intended for use outside the class, called *public methods*, and those intended for use by the class and those classes you derive from

it, called *protected methods*. *Private methods* are usually only called inside the object itself by other parts of the object. In the Previously-mentioned refrigerator example, for instance, the thermostat may call an entirely internal method named `start_compressor` when it's time to get cold. Once you have an object that supports methods, you can use that object's methods. In the following example, we use the `calculate()` method to work with the two values in `operand1` and `operand2` and store the result of the calculation in `result`:

```
Calculator calc = new Calculator();
result = calc.calculate(operand1, operand2);
```

Java supports two types of methods: class methods and instance methods. Instance methods, like the `calculate()` example here, are invoked on objects (i.e., objects are instances of a class). Class methods, on the other hand, are invoked on a class. For example, the `java.lang.Math` class has a class method named `sqrt()` that calculates a square root, and you can use it like this (no object is needed):

```
public class App {
  public static void main(String[] args) {
        double value = 4, sqrt;
        sqrt = Math.sqrt(value);
        System.out.println("The square root of " + value + " = " + sqrt); }
}
```

Here's what you see when you run this code:

```
C:\>java App
The square root of 4.0 = 2.0
```

You'll learn how to create both instance and class methods in this chapter. There's one more object-oriented concept to master before we get to the code—inheritance.

Inheritance

Inheritance is one of the formally defining aspects of Object-Oriented Programming. Using inheritance, you can *derive* a new class from an old class, and the new class will *inherit* all the methods and member data of the old class. The new class is called the *derived class*, and the original class is called the *base class*. The idea here is that you add what you want to the new class to give it more customized functionality than the base class.

For example, if you have a class named `Vehicle`, you might derive a new class named `Car` from `Vehicle` and add a new method called `horn()` that prints "beep" when called. In that way, you have created a new class from a base class and have augmented that class with an additional method. Inheritance is an important topic in Java because you can use the huge class libraries available in Java by deriving your own classes from them. You'll see how to use object-oriented inheritance in the next chapter.

Exception Handling

An *exception* refers to an event that occurs when a program is not being executed in the normal flow, i.e., not according to the instructions. Each type of exception is represented by a different exception class:

- `IllegalArgumentException`—It occurs when passing an incorrect argument to a method.
- `InputMismatchException`—It occurs when the console input doesn't match the data type expected by a method of the `Scanner` class.
- `ArithmeticException`—It occurs when an illegal type of arithmetic operation is performed.
- `IOException`—It occurs when the I/O encountered an unrecoverable I/O error.
- `ClassNotFoundException`—It occurs when a necessary class couldn't be found.

They are just some exceptions that you'll find as you read on. Each of these topics is important, and we'll take a look at them in overview now.

Exception handling is the way that Java handles runtime errors. The basis of exception handling is the `try` block, in which you place code that can cause exceptions. In fact, you'll see this throughout the book, such as in this example, where an operation using an `InputStreamReader` can generate an exception (called throwing an exception), which is then caught and handled in a `catch` block:

```
import java.io.*;
class App {
  public static void main(String args[]) {
```

```
            try {
                    int character;
                    InputStreamReader reader = new
                    InputStreamReader(System.in);
                    while ((character = reader.read()) != -1) {
                            System.out.print((char) character); }   }
            catch (IOException e) { System.out.println("Error occurred!"); }
      }
   }
```

You'll know more on `InputStreamReader` in later chapters. But for now, it is important for you to know that it reads bytes and translates them into characters according to a specified character encoding.

Using Java's built- in I/O classes, you can open a file for reading, outputting each line to the screen until the end of the file is reached. Such I/O operations do generate errors, so these I/O operations are typically placed in `try` blocks, because they're error-prone—Java will usually insist that you use a `try/catch` block (and if you don't, it'll ask you to do so when you try to compile your code).

Here, the sensitive operation is enclosed in a `try` block; and if an exception occurs, it's caught in the `catch` block. You can do various things in the `catch` block, such as attempt to recover from the exception, ignore it, inform the user, and so on. You'll see the options in this chapter.



It's a good idea to handle exceptions yourself in any program you release for general use because if you don't, Java passes the exception on to the default exception handler, which prints out its internal stack and terminates your program.

Debugging

Exception handling lets you work with runtime errors, but there's another kind of error: logic errors in your code (also known as *bugs*). Java provides a debugging tool— the `jdb` tool—that lets you single-step through your code, set *breakpoints* in your code that halt execution when you reach them, examine the variables of a program while it's executing, and more. We'll take a look at the `jdb` tool in this chapter.

That's it for the overview of what's in this chapter. There's a great deal of material here, and it's time to turn to the "Immediate Solutions" section and master OOP in detail. All this material is essential for Java programming, so keep digging into it until you've made it your own.

Immediate Solutions

Declaring and Defining Classes

A Novice Programmer is excited and says, "I have done it! I have created an object—it worked!" "Fine," you say approvingly, "now how about creating a class?" The NP says, "How does that work?"

There are two parts for setting up a class in Java: the class declaration and the class definition. The declaration tells Java what it needs to know about the new class. The general form of a class declaration is as follows:

```
[access] class classname [extends ...] [implements ...] {
  //class definition goes here.
}
```

The actual implementation of the class is called the *class definition*, and it makes up the body of the class declaration, which you can see in the preceding sample code. The general form of a class declaration and definition is as follows:

```
access class classname [extends ...] [implements ...] {
  [access] [static] type variable1;
  . . .
  [access] [static] type variableN;
  [access] [static] returntype method1 (parameter_list) { . . . }
  . . .
  [access] [static] returntype methodN (parameter_list) { . . . }
}
```

The `static` keyword in the general form turns variables into class variables and methods into class methods (as opposed to instance variables and methods), as you'll see later. The *access* term specifies the accessibility of the class or class member to the rest of the program, and it can be `public`, `private`, or `protected`. There's also a default access if you don't specify an access type; you'll learn about this in the next few pages. You use the `extends` and `implements` keywords with inheritance, which you'll see in the next chapter.

An example will make this all clear. To start, we'll just create a very simple class named `Printer` that defines one method `print()`. When we call the `print()` method, it displays the message "Hello from Java!" on the console. The Printer class appears as follows:

```
class Printer {
  public void print() { System.out.println("Hello from Java!"); }
}
```

Now we can make use of the `print()` method in other classes, as in this example, where we're creating a new object of the `Printer` class by using the `new` operator and that object's `print()` method in an application named *App*:

```
class Printer {
  public void print() { System.out.println("Hello from Java!"); }
}
public class App {
  public static void main(String[] args) {
        Printer printer1 = new Printer();
        printer1.print(); }
}
```

Now we put this code in a file (`App.java`), compile it, and then run it as follows:

```
C:\>java App
Hello from Java!
```

Note that we're declaring and defining two classes, `Printer` and `App`, in the same file. Only one class can be declared public in one file, and that's `App` in this case. The file itself must be named after that class, which means the containing file must be `App.java`. However, you can have as many private or protected classes as you like in the file (and Java will create separate `.class` files for them when you compile the file).

You can also divide this example into two files, one for each class. Here's `Printer.java` that displays a message:

```
package printer;
class Printer {
  public void print() { System.out.println("Hello from Java!"); }
}
```

The new `App.java` is as follows:

```
import printer.Printer;
public class App {
  public static void main(String[] args) {
        Printer printer1 = new Printer();
        printer1.print(); }
}
```

Note that we had to import the Printer class to be able to use it; see Chapter 1 for more on importing classes.

Declaring and Creating Objects

The Novice Programmer (NP) appears to discuss Object-Oriented Programming. "I know all about objects now," the NP says, "only …" "Only what?" you ask. "Only that I don't know how to actually *create* an object in a program."

You need to declare an object before you can use it. You can declare objects the same way you declare variables of simple data types, but using the class as the object's type. Also, you can use the `new` operator to create objects in Java. Let's look at an example by using the Java `String` class.

TIP

Actually, you don't need to declare an object in all cases before using it. In some cases, Java creates an object for you automatically, as is the case with string literals, which Java treats like String objects. This means that expressions such as "Hello from Java!".length() are valid.

To start, we'll declare a new object, `s1`, of the `String` class:

```
public class App { public static void main(String[] args) {
        String s1;
        . . . }
}
```

Although declaring a simple variable creates that variable, declaring an object doesn't create it. To actually create the object, we can use the `new` operator using this general form in which we're passing parameters to the class's constructor:

```
object = new class([parameter1 [, parameter2...]]);
```

The `String` class has several constructors, as you saw in Chapter 2. You can pass quoted strings to one of the `String` class's constructors; therefore, you could create the new object, `s1`, like this:

```
public class App { public static void main(String[] args) {
        String s1;
        s1 = new String("Hello from Java!");
        . . . }
}
```

Now the new object, `s1`, exists and is ready for use. For example, to convert all the characters in `s1` to lowercase, you can use the `String` class's `toLowerCase()` method, like this:

```
s1.toLowerCase()
```

The declaration and creation steps can be combined together into one step. Here's an example in which we're declaring a new `String` object, `s2`, and creating it with the `new` operator, all in one line:

```
public class App {
  public static void main(String[] args) {
        String s1;
        s1 = new String("Hello from Java!");
        String s2 = new String("Hello from Java!");
        . . . }
}
```

Classes often have several different constructors, each of which can take a different data specification (i.e., different parameter types and number of parameters; the Java compiler knows which constructor you want to use by noting how the types of the parameters are used and how many parameters are there). In object-oriented terms, these constructors are *overloaded*, and we'll cover overloading in this chapter. For example, the `String` class's constructor is overloaded to take character arrays as well as text strings, so we can create a new object, `s3`, using a character array, like this:

```
public class App {
  public static void main(String[] args) {
        String s1;
        s1 = new String("Hello from Java!");
        String s2 = new String("Hello from Java!");
        char c1[] = {'H', 'i', ' ', 't', 'h', 'e', 'r', 'e'};
        String s3 = new String(c1);
        . . . }
}
```

Sometimes classes will have methods that return objects, which means they'll use the `new` operator internally (and you don't have to). Here's an example, using the `valueOf()` method of the `String` class, in which we convert a `double` into a `String` object:

```
public class App {
  public static void main(String[] args) {
        String s1; s1 = new String("Hello from Java!");
        String s2 = new String("Hello from Java!");
        char c1[] = {'H', 'i', ' ', 't', 'h', 'e', 'r', 'e'};
        String s3 = new String(c1);
        double double1 = 1.23456789;
        String s4 = String.valueOf(double1);
        . . . }
}
```

In addition, you can assign one object to another, as we have done here:

```
public class App {
  public static void main(String[] args) {
        String s1;
        s1 = new String("Hello from Java!");
        String s2 = new String("Hello from Java!");
        char c1[] = {'H', 'i', ' ', 't', 'h', 'e', 'r', 'e'};
        String s3 = new String(c1);
        double double1 = 1.23456789;
        String s4 = String.valueOf(double1);
        String s5;
        s5 = s1;
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s3);
        System.out.println(s4);
        System.out.println(s5); }
}
```

Internally, what's really happening is that the object reference in `s1` is copied to `s5`. What this means in practice is that `s1` and `s5` refer to the *same* object. That's important to know, because if you change the instance data in `s1`, you're also changing the instance data in `s5`, and vice versa. If two variables refer to the same object, be careful—multiple references to the same object can create some extremely hard-to-find bugs. This usually happens when you think you're really dealing with different objects.

At the end of the preceding code, we print out all the strings we've created. Here's what appears when the program is run:

```
C:\>java App
Hello from Java!
Hello from Java!
Hi there
1.23456789
Hello from Java!
```

That's how to declare and create objects—much the same way you declare and create simple variables, with the added power of configuring objects by passing data to a class's constructor. It's time to start creating your own classes, and we'll start that process in the next solution.

Creating Instance Variables

"Hmm," says the Novice Programmer, "I want to create a class to store data, and I'm all set except for one small thing." "Yes?" you ask. "How do I store data in classes?" the NP asks.

You can store data in classes in one of two ways—as instance variables or as class variables. Instance variables are specific to objects; if you have two objects (i.e., two instances of a class), the instance variables in each object are independent of the instance variables in the other object. On the other hand, the class variables of both objects will refer to the same data and therefore hold the same value. Let's take a look at instance variables first. Here's how you declare instance variables in a class:

```
access class classname [extends ...] [implements ...] {
  [access] type instance_variable1;
  . . .
  [access] type instance_variableN;
}
```

Here's an example in which we'll create a class named `Data` that holds a String instance variable named `data_string`, which in turn holds the text "`Hello from Java!`":

```
class Data {     public String data_string = "Hello from Java!"; }
```

Now we can create an object, named `data`, of the `Data` class in `main` and refer to the instance variable `data_string` in `data` as `data.data_string`. Here's what this looks like in code:

```
class Data {     public String data_string = "Hello from Java!"; }
public class App {
  public static void main(String[] args) {
        Data data = new Data();
        String string = data.data_string;
        System.out.println(string); }
}
```

As you can see, you can access the public instance variables of an object with the dot operator. However, remember that one of the motivations behind Object-Oriented Programming is to keep data private. We'll take a look in more detail on this in the next topic.

Setting Variable Access

"Hey," says the Novice Programmer, "I thought objects were supposed to encapsulate data in a private way—how come that Johnson has been able to access the data inside my objects?" "Because you used the wrong access specifier for your data," you say.

You can use an access specifier—called *`access`* in the following code—to set the visibility of a class's data members as far as the rest of the program is concerned:

```
access class classname [extends ...] [implements ...] {
  [access] [static] type instance_variable1;
  . . .
  [access] [static] type instance_variableN;
}
```

The access specifier *`access`* can contain three possible values: `public`, `private`, and `protected`. You can access a `public` data member from anywhere in your program, a `private` data member only in the current class, and `protected` data member in the current class, other classes in the same package, and its subclasses. If no access specifier is used, the member has default access, which means that the class member is accessible in the current class, subclasses in the same package, and other classes in the same package. You will learn more about creating subclasses in Chapter 5.

You can find the details in Table 4.1.

For example, if we wanted to make the instance variable `data_string` private to the `Data` class created in the previous topic, we can declare it `private`, like this:

```
class Data {     private String data_string = "Hello from Java!"; }
public class App {
  public static void main(String[] args) {
        Data data = new Data();
        String string = data.data_string;
        System.out.println(string); }
}
```

Now if we try to access the `data_string` instance variable in another class, as we did previously in the `App` class, the Java compiler will object:

```
C:\>javac App.java
App.java:5: error: data_string has private access in Data
            String string = data.data_string;
                                 ^
1 error
```

Table 4.1: Scope by access specifier (x means "in scope")

Location	Private	No modifier	Protected	Public
Same class	x	x	x	x
Subclass in the same package	.	x	x	x
Non-subclass in the same package	.	.	x	x
Subclass in another package	.	.	x	x
Non-subclass in another package	.	.	.	x

Creating Class Variables

"Uh-oh," says the Novice Programmer, "I have a new class named `Counter`, and I need to keep track of the total count in a variable named `counter` for *all* objects of that class. Now what? I'm sunk." "You're not sunk," you say. "You just need to use a class variable."

All objects of that class share the value in a class variable, which means it will be the same for all objects. You declare a variable as static with the `static` keyword (which really specifies the way the value is stored, as static data, as opposed to other variables, which are stored dynamically on stacks):

```
access class classname [extends ...] [implements ...] {
  [access] static type variable1;
  . . .
  [access] static type variableN;
}
```

Here's an example in which we create a class named `Data` with a class data variable named `intdata`:

```
class Data {public static int intdata = 0; }
```

Now we can create two objects of the `Data` class: `a` and `b`. When we set the `intdata` variable for `a` to `1`, we find that the `intdata` variable for `b` is also set to 1, as you can see here:

```
class Data {     public static int intdata = 0; }
public class App {
  public static void main(String[] args) {
        Data a, b;
        a = new Data(); b = new Data();
        a.intdata = 1;
        System.out.println("The value of b.intdata = " + b.intdata); }
}
```

Here's the result of this code:

```
C:\>java App
The value of b.intdata = 1
```

If you need to perform some calculation to initialize static variables, you can do so in a static code block, which you label with the `static` keyword; that code is executed just once, when the class is first loaded:

```
class Data {
  public static int intdata = 1;
  public static int doubledintdata;
```

```
      static { doubledintdata = 2 * intdata; }
   }
   public class App {
      public static void main(String[] args) {
            System.out.println("The value of doubledintdata = " +
            Data.doubledintdata); }
   }
```

Here's the result of this code:

```
C:\>java App
The value of doubledintdata = 2
```

Creating Methods

"OK," says the Novice Programmer, "I have got instance variables down now. Is there anything more to learn about classes?" "Plenty," you say. "Pull up a chair and we'll talk about creating methods."

Till now, we have used a lot of methods; the most commonly used is the method `println()` in the statement `System.out.println()` to display the output at the console.

A *method* is a code block that you can transfer control to so as to execute that code. Here's how you create methods in a class:

```
access class classname [extends ...] [implements ...] {
   [access] [static] returntype method1 (parameter_list) { . . . }
   . . .
   [access] [static] returntype methodN (parameter_list) { . . . }
}
```

To declare and define a method, you use an access specifier (see the next topic), specify the return type of the method if you want it to return a value (such as `int`, `float`, an object type, or `void` if the method doesn't return any value), give the method's name and place the list of the parameters you intend to pass to the method after that name. The actual method body is written after the method's declaration and contains the instructions that are executed when the method is called. Let's get to an example. In fact, you've already seen one earlier in this chapter—the `Printer` class. In that example, we added a public method named `print()` to the `Printer` class, created an object of the `Printer` class, and called the `print()` method, like this:

```
class Printer {
   public void print() { System.out.println("Hello from Java!"); }
}
public class App {
   public static void main(String[] args) {
         Printer printer1 = new Printer();
         printer1.print(); }
}
```

In this case, the `print()` method takes no parameters and returns no value, but we still use parentheses after the method's name—this is mandatory when you are defining or calling a method in Java (it's mandatory because that's how the Java compiler knows `print()` is a method and not a data member). Here's the output of this code:

```
C:\>java App
Hello from Java!
```

There's a lot to know about creating methods in Java, so we're going to elaborate on the process over the next few topics. One of the most important aspects of methods is that you can make them purely internal to an object, in keeping with the object-oriented concept of encapsulation—and that's where we'll start.

Setting Method Access

"That Johnson," the Novice Programmer says, "has been using the internal method in my objects, even though I clearly named the method `internal_use_only`. Isn't there anything stronger I can use to keep that silly Johnson out?" "Yes," you say, "you can use a stronger access specifier." "Great!" says the NP.

You can add an access specifier to the methods in a class, like this (where *`access`* is the access specifier):

```
access class classname [extends ...] [implements ...] {
   [access] [static] returntype method1 (parameter_list)
```

```
    { . . . }
    . . .
    [access] [static] returntype methodN (parameter_list)
    { . . . }
}
```

The access specifier *access* can contain three possible values: `public`, `private`, and `protected`. You can find the details of them in Table 4.1.

Here's an example in which we add a `private` method to the `Printer` class developed over the last few topics. This method may only be called from other methods in the `Printer` class, like this:

```
class Printer {
   public void print() { internal_use_only(); }
   private void internal_use_only () {
         System.out.println("Hello from Java!"); }
}
public class App {
   public static void main(String[] args) {
         Printer printer1 = new Printer(); printer1.print(); }
}
```

When you call the `Printer` class's `print()` method, it makes use of the `internal_use_only()` method, which is inaccessible outside the object, to do the actual printing.

Here's the result of this code:

```
C:\>java App
Hello from Java!
```

Making methods private or protected is often a good idea because it reduces or controls the method's accessibility from the rest of the code.

Passing Parameters to Methods

The company's Customer Support Specialist gives you a call and says, "We have an issue." "What issue would that be, CSS?" you ask. "Your `Printer` class prints out a message, but customers are complaining because they want to set the message that's printed out." "No problem," you say. "I'll set the `print()` method up to their expectations, i.e., by accepting parameters."

When you declare a method, you can specify a comma-separated list of parameters that you want to pass to that method in the parentheses following the method's name:

```
[access] [static] returntype method1 ([type parameter_name1 [, type
parameter_name1...]])
{ . . . }
```

The values passed to the method will then be accessible in the body of the method; using the names you've given them in the parameter list. Here's an example in which we pass the string to print to the `print()` method. We declare the method, like this, so that Java knows it will accept one parameter—a String object named `s`:

```
class Printer {
   public void print(String s) { . . . }
}
```

Now we can refer to the String object passed to the `print()` method as `s` in the body of the method:

```
class Printer {
   public void print(String s) { System.out.println(s);  }
}
public class App {
   public static void main(String[] args) {
         Printer printer = new Printer();
printer.print("Hello again from Java!");

}
}
```

Here's the result of this code:

```
C:\>java App
Hello again from Java!
```

If you have more than one parameter to pass, you can specify multiple parameters in the parameter list, separated by commas:

```
class Calculator {
  int additems(int op1, int op2) {
        int result = op1 + op2;
        . . . }
}
```

You can call methods using literals, variables, arrays, or objects, like this:

```
calc.additems(1, int1, array1, obj1)
```

You should note that when you pass a simple variable or literal to a method, the value of the variable or literal is passed to the method—this process is called *passing by value*. On the other hand, when you pass an object or array, you are really passing a *reference* to that object or array (in fact, when you store an array or object in a variable, what you're really storing is a reference to the array or object). For that reason, the code in the called method has direct access to the original array or object, not a copy of it. Therefore, if that code changes some aspect of the array or object, such as an element in the array or a data member of the object, the original array or original object is changed. We'll take another look at this in detail in the topics, "Passing Objects to Methods" and "Passing Arrays to Methods", later in this chapter.

Command-Line Arguments Passed to **main() method**

A special array is passed as a parameter to the `main()` method in applications—an array of `String` objects that holds the command-line arguments the user specified when starting Java. For example, suppose you started an application this way:

```
C:\>java App Kogent Learning Solutions
```

In this case, the first element of the array passed to `main()` will hold "Kogent", the second "Learning", and the third "Solutions". Here's an example showing how this works; this application will print out all the command-line arguments passed to it by looping over the `String` array passed as a parameter to the `main()` method:

```
public class App {
  public static void main(String[] args) {
        System.out.println("Command line arguments...");
        for(int loop_index = 0; loop_index < args.length; loop_index++) {
              System.out.println("Argument" + loop_index + " = "
              + args[loop_index]); }
  }
}
```

Here's how we might put this application to work:

```
C:\>java App Kogent Learning Solutions
Command line arguments...
Argument 0 = Kogent
Argument 1 = Learning
Argument 2 = Solutions
```

Returning Values from Methods

The Novice Programmer is back and says, "Well, there's another problem. The Big Boss wants me to create a `Calculator` class that can perform mathematical operations. I can accept passed parameters in the methods of that class, but …" "Yes?" you ask. "I can't send any results back from the methods of the class after I've done the math." "Ah," you say, "use the `return` statement."

The `return` statement is used in a method for returning a value from the method, and you indicate what the return type of the method is when you declare the method:

```
[access] [static] returntype method1 ([type parameter_name1 [, type
parameter_name1...]])
{
      . . .
      return value;
}
```

The return type can be any type that Java recognizes—for example, `int, float, double`, the name of a class you've defined, `int[]` to return an integer array, or `float[]` to return a `float` array. Here's an example in which the class `Calculator` has a method named `additems()` that takes two integer parameters, adds them, and returns the result. Here's how we declare `additems()`:

```
class Calculator {
  int additems(int op1, int op2) { return result; }
}
```

Here's how we return the sum of the values passed to `additems()`, using the `return` statement:

```
class Calculator { int additems(int op1, int op2) { return op1 + op2; } }
```

Here's how we put the `Calculator` class to work in a program:

```
class Calculator { int additems(int op1, int op2) { return op1 + op2; } }
  public class App {
        public static void main(String[] args) {
              Calculator calc = new Calculator();
System.out.println("additems(2,2) = " + calc.additems(2,2)); }
  }
```

Here's the result of this application:

```
C:\>java App
additems(2,2) = 4
```

Creating Class Methods

"Wow?" says the Novice Programmer. "I have created my new `Calculator` class with a terrific method in it named `additems()`, but why do I have to go to the trouble of creating an object of that class before I can use the `additems()` method? Can't I just call that method directly?" "You can," you say, "if you make `additems()` a *class* method instead of an *instance* method."

To make a method into a class method, use the `static` keyword:

```
class Calculator { static int additems(int op1, int op2) { return op1 + op2; } }
```

Now in code, you can call the `additems()` method directly by using the class name, without creating an object at all. Here's an example:

```
public class App {
  public static void main(String[] args) {
        System.out.println("additems(2,2) = " + Calculator.additems(2,2));   }
}
```

Here's the result of this code:

```
C:\>java App
additems(2,2) = 4
```

You can also use a class method the usual way—as a method of an object:

```
class Calculator { static int additems(int op1, int op2) { return op1 + op2; } }
  public class App {
        public static void main(String[] args) {
              Calculator calc = new Calculator();
              System.out.println("additems(2,2) = " + calc.additems(2,2)); }
  }
```

It's worth noting that the `main()` method in an application is declared `static` because Java must call it before an object actually exists.

If you declare a method `static` (this includes the `main()` method in any application), it can only call other static methods and can only access static data. Also, it cannot use the `this` and `super` keywords, which refer to the current object and the parent object of the current object, respectively, as you'll see in this and the next chapter. Note, in particular, that you can't refer to instance data in a `static` method.

TIP

So how do you call non-static methods from main? You do this as we have been doing it throughout the book—you create an object of some other class in main and call the methods of that object.

Creating Data Access Methods

"Again that Johnson," the NP says, "is fiddling around inside my code's objects. But this time, I can't declare everything `private` because the rest of the code needs access to the data member in question. What can I do?" "You can set up a data access method," you say, "and restrict access to your data members in a well-defined way." "That'll show that Silly Johnson!" the NP says.

You can restrict access to the data in your objects by using data access methods, which must be called to fetch the data. Here's an example in which we have got a private `String` data member called `data_string`:

```
class Data {
   private String data_string = "Hello from Java!";
   . . .
   }
```

We can provide access to this private data member with two methods—`getData()` and `setData()`. The `getData()` method just returns the value in the private variable `data_string`, like this:

```
class Data {
   private String data_string = "Hello from Java!";
   public String getData() { return data_string; }
}
```

However, the `setData()` method restricts access to the internal data; in particular, we'll write this method so that the calling code can only set the internal data to a new string if the length of that string is less than 100 characters. Here's how this looks:

```
class Data {
   private String data_string = "Hello from Java!";
   public String getData() { return data_string; }
   public void setData(String s) {
                  if (s.length() < 100) { data_string = s; }
   }
}
```

Now we can use the `getData()` method to get the internal string and the `setData()` method to set it to a new string. Here's an example that shows how to use `getData()`:

```
public class App {
   public static void main(String[] args) {
          System.out.println((new Data()).getData()); }
}
```

Here's the result of this code:

```
C:\>java App
Hello from Java!
```

Using data access methods to grant access to the internal data in your objects is a good idea. By using these methods, you can control the interface to the data, thus, blocking operations you consider are illegal.

Creating Constructors

"Hmm," says the Novice Programmer, "I know I can use constructors to initialize the data in an object, such as the `String` class's constructors that I use to set the text in a string, but…" "Yes?" you ask. "But how can *I* create constructors for my own classes?" the NP asks.

Creating a constructor for a class is easy; you just add a method to a class with the same name as the class, with or without any access specifier and having no return type. Here's an example in which we add a constructor that takes no parameters to the `Printer` class we've developed in this chapter. This constructor is called when an object is created of the `Printer` class; and in this case, it initializes the internal data `data_string` to "Hello from Java!" (Note that we still need the parentheses after the constructor name when declaring it, even though it doesn't take any parameters):

```
class Data {
   private String data_string;
   Data() { data_string = "Hello from Java!"; }
   public String getData() { return data_string; }
}
```

```
public class App {
  public static void main(String[] args) {
        System.out.println((new Data()).getData()); }
}
```

Here's what you see when you run this program:

```
C:\>java App
Hello from Java!
```

This constructor is a particularly simple one because it doesn't take any parameters. We'll enable the constructor to take parameters in the next topic.

Passing Parameters to Constructors

"OK," says the Novice Programmer, "Java's gone wacky again. I set up a constructor for my new class, but the object isn't actually initialized with the data I want." "Hmm," you say, "did you pass any data to the constructor?" "Uh-oh," says the NP.

You can pass data to constructors just as you can to other methods. Here's an example, using the `Printer` class from the previous topic, in which we pass the string to print out to the `Printer` class's constructor:

```
class Data {
  private String data_string;
  Data(String s) { data_string = s; }
  public String getData() { return data_string; }
}
public class App {
  public static void main(String[] args) {
        System.out.println((new Data("Hello from Java!")).getData()); }
}
```

Here's the result of this code:

```
C:\>java App
Hello from Java!
```

And that's all it takes—passing parameters to a constructor works the same as passing parameters to any method.

A Full Class Example

Let's design a class implementing constructors and some member methods for it. We are designing a class to represent a complex number. A complex number has two parts: one is real and another is imaginary. A complex number (Z) is represented as `Z=a+bi`, where 'a' is real part and 'b' is imaginary part. We can define constructors to initialize complex object with given set of real and imaginary values. Further, there are two methods which are required here. One is to print a complex number and another is to add two complex numbers. See the following code which creates a class `Complex`. This class has two constructors: one zero argument constructor and another two argument constructor. The `Complex` class's `add(Complex)` method adds two complex numbers and returns a complex number.

One good question which comes in mind here is that we are passing one complex number in `Complex` class's `add(Complex)` method to which it adds two complex numbers. Where is second complex number? The answer is simple. The object for which this method is invoked is taken as second complex to add with.

```
public class Complex {
  int real;
  int imag;
  public Complex() {
        real=0;
        imag=0; }
  public Complex(int r, int i) {
        real=r;
        imag=i; }
  void show_complex(){
        System.out.print(real+"+"+imag+"i"); }
  Complex add(Complex obj) {
```

```
            Complex temp=new Complex();
            temp.real=this.real+obj.real;
            temp.imag=this.imag+obj.imag;
            return temp; }
      public static void main(String args[]) {
            Complex z1=new Complex(2,4);
            Complex z2=new Complex(4,5);
            System.out.print("\nFirst Complex number Z1 = ");
            z1.show_complex();
            System.out.print("\nSecond Complex number Z2 = ");
            z2.show_complex();
            Complex z3=z1.add(z2);
            System.out.print("\nAdding Z1 and Z2...");
            System.out.print("\nZ1 + Z2 = ");
            z3.show_complex(); }
   }
```

Here's what this program looks like at work:

```
C:\>java Complex
First Complex number Z1 = 2+4i
Second Complex number Z2 = 4+5i
Adding Z1 and Z2...
Z1 + Z2 = 6+9i
```

Understanding Variable Scope

"Hmm," says the Novice Programmer, "I have defined a swell new variable named `the_answer` in a method named `get_the_answer()`, and I was trying to use that variable in a method named `get_a_clue()`, but Java claims the variable is undefined!" "Hmm," you say, "sounds like a question of variable scope—you can't use variables declared in one method in another method." "You can't?" the NP asks.

The *scope* of a variable is made up of the parts of the program in which that variable can be used in your code; and as you can see from the Novice Programmer's plight, scope is an important concept to understand. Java defines three main scopes—class-, method-, and code block-level. If you define a data member in a class, that data member is available throughout the class, and possibly beyond, as you've seen with the `private`, `public`, and `protected` access specifiers.

The scope of a method starts when the flow of execution enters the method and ends when the flow of execution leaves the method. Variables declared in the method are visible in the method only. The data members of the class are also visible in the class's methods, as are the parameters passed to those methods.

You can also define a local scope for variables using code blocks, because you can declare variables inside code blocks. The variables you declare inside a code block will be visible only in that code block and in any code blocks contained within the code block.

The easiest way to bear all this in mind is to know that non-static variables declared in a code block bounded by curly braces are created and stored on the local stack when you enter that code block, and they're destroyed when you leave the code block (which is why they're called *dynamic variables*). Static variables, on the other hand, are stored in the program's own data allocation, not on any stack, which is why they don't go out of scope. They're as close to global variables (i.e., program-wide variables) as Java permits. Here's an example showing the various levels of scope (class, method, and code block):

```
class Scope {
   int int1 = 1; //visible to all code in the class.
   public void method(int int2) { //visible to all code in this method.
         int int3 = 3; //visible to all code in this method.
         if(int1 != int2) {
               int int4 = 4; //visible only in this code block.
               System.out.println("int1 = " + int1
               + " int2 = " + int2 + " int3 = " + int3
               + " int4 = " + int4); }
   }
}
```

```
public class App {
  public static void main(String[] args) {
        Scope s = new Scope();
        s.method(2); }
}
```

Here's what you see when this code runs:

```
C:\>java App
int1 = 1 int2 = 2 int3 = 3 int4 = 4
```

Using Recursion

The Novice Programmer comes in still shaking with laughter and says, "You'll never believe what the Programming Correctness Czar just told me—in C++, methods can call themselves!" "It's the same in Java," you reply. "Huh?" the NP says.

Each time you call a method in Java, Java allocates new space on its internal stack for all the variables in the method, which means there's no reason you can't call the same method again—a new set of variables will be allocated on the stack automatically. What's more, a method can call itself in Java—this is a technique called *recursion*.

The classic recursion example is to calculate a factorial, so we'll implement it here. To calculate the factorial of positive integer *n*, you calculate the following:

```
n! = n * (n - 1) * (n - 2) ... * 2 * 1
```

This process lends itself to recursion easily because each stage of the recursion can calculate one multiplication in which it multiplies the number it has been passed by the factorial of the number minus 1. When the number has finally been reduced to 1 through successive calls, the method simply returns, and control comes back through the successive stages, performing one multiplication at each stage until all nested calls have returned and you have the factorial.

Here's what this looks like in code:

```
class Calculator {
  public int factorial(int n) {
                if (n == 1) { return n; }
        else { return n * factorial(n - 1); }
  }
}
public class App {
  public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("6! = " + calc.factorial(6)); }
}
```

Here's what this program looks like at work:

```
C:\>java App
6! = 720
```

In practice, you probably won't use recursion too often, but it's good to know it's available.

Garbage Collection and Memory Management

"Say," the NP says, "I just thought of something—you allocate new memory with the `new` operator, but how do you get rid of it when it's no longer needed? Is there an `old` operator?" "Nope," you say, "Java does all that for you."

In some languages, such as C++, you use the `new` operator to allocate new memory, and then you use the `delete` operator to get rid of it when you don't need it anymore. However, Java does not have a `delete` operator. So, how do you get rid of allocated memory when it's no longer needed? In Java, you have to rely on a built-in process called *garbage collection*. This process occurs automatically, although you can't predict when it will happen. Java, itself, will dispose off allocated memory that no longer has any references to it. To make garbage collection happen, you can set any references to an item to `null` (although doing so still does not let you predict when, if ever, garbage collection will happen when your program is executing). Here's an example

in which we're just creating a new object and then setting its variable to `null`. Because there are no remaining references to the object, the garbage collection process will deallocate it sooner or later. Here's the code:

```
class Data {
  public int intdata = 0;
         Data() { intdata = 1; }
}
public class App {
  public static void main(String[] args) {
         Data d = new Data(); //some code...
         d = null; } //some additional code...
  }
```

Here's the thing to remember from this example: When you are done with a data item—including objects and arrays—that you've allocated with the `new` operator, you can set its references to `null`, and if Java needs more memory, it'll start the garbage collection process. However, you have to be careful to avoid circular references.

TIP

If you are familiar with C++, you may be wondering where pointers are in Java, and the answer is that it doesn't have them. The designers of Java omitted pointers for security reasons– to make sure programmers couldn't access memory beyond legal limits. Instead of pointers, Java uses references, which act very much like pointers behind the scenes. When you create a new object, you get a reference to that object, and when you use that reference, Java de-references it for you automatically. That's how it works in Java—Java handles references (pointers) for you automatically.

Avoiding Circular References

Garbage collection—the disposing of memory items that no longer have any references in your program—happens automatically. However, you should watch out for circular references in which one object has a reference to another and the second object has a reference to the first. When you get rid of any references to these objects in your program, each object still has an internal reference to the other, which means garbage collection can't happen on either object. Worse yet, because there are no external references to either object, you can't reach either object to try to change the situation. Both objects will sit in memory, taking up valuable resources, until your program ends.

Here's a sample program showing what we mean—in this case, class `A` has an internal reference to an object of class `B`, and class `B` has an internal reference to an object of class `A`. When the code in `main()` sets the reference it has to one of these objects to `null`, these objects will continue to sit in memory until the program ends. Here's the code:

```
class A {
  B b1;
         A() { b1 = new B(); }
}
class B {
  A a1;
  B() { a1 = new A(); }
}
public class App {
  public static void main(String[] args) {
         A obj = new A();
         obj = null; } //inaccessible circular references now exist!
  }
```

There's only one way to avoid this, and that's to get rid of circular references before cutting them adrift. In practice, this usually means setting an object's references to other objects to `null` before setting the reference to the object, itself, to `null`. Sometimes it is possible to do this in the `finalize()` method (see the next topic for the details). While we are discussing memory management, it's also worth noting that you do have some control over memory allocation as a whole—see the `-J` command-line option in Chapter 1, which lets you set the total amount of memory allocated when a program runs. In general, though, Java handles the memory management in your programs.

Garbage Collection and the **finalize()** Method

"Hmm," says the Novice Programmer, "so Java has a garbage collector that removes items from memory that are no longer referenced. Is there anything more I should know about this process?" "One thing," you say. "The garbage collector calls a special method named `finalize()` in your object if that method exists, and you can use this method for last-minute cleanup."

When an object is being "garbage collected" (see the previous topic), the garbage collector will call a method named `finalize()` in the object, if it exists. In this method, you can execute cleanup code, and it's often a good idea to get rid of any references to other objects that the current object has in order to eliminate the possibility of circular references (also covered in the previous topic).

Here's an example showing how this looks in code:

```
class Data {
  public int intdata = 0;
  SuperGiantSizeClass sgsc;
  Data() { intdata = 1; sgsc = new SuperGiantSizeClass(100000000); }
  protected void finalize() { sgsc = null; }
}
public class App {
  public static void main(String[] args) {
        Data d = new Data();
        d = null; }
}
```

Overloading Methods

"I'm still working on my new program, *SuperDuperMathPro*," says the Novice Programmer, "and I have a great class named `Calculator` with a method named `additems()` that adds two numbers. I'd also like to add three numbers together, though, I guess I'll have to write a new method." "Not at all," you say. "You can overload the `additems()` method to handle either two *or* three operands." "How's that?" the NP asks.

Method overloading is an object-oriented technique that lets you define several different versions of a method, all with the same name, but each with a different parameter list. When you use an overloaded method, the Java compiler will know which one you mean by the number and/or types of the parameters you pass to the method, finding the version of the method with the right parameter list.

Let's take a look at an example. To overload a method, you just define it more than once, specifying a new parameter list each time. Each parameter list must be different from every other one in some way, such as the number of parameters or the type of one or more parameters. We'll create an example the Novice Programmer was worried about here. First, we add a version of the `additems()` method to the `Calculator` class that will handle two operands:

```
class Calculator {
  int additems(int op1, int op2) {return op1 + op2; }
        . . .
}
```

Then we add another version of the same method that will take three operands:

```
class Calculator {
int additems(int op1, int op2) { return op1 + op2; }
        int additems(int op1, int op2, int op3) { return op1 + op2 + op3; }
}
```

Now we can use both methods in code, like this:

```
public class App {
  public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("additems(2,2) = " + calc.additems(2,2));
        System.out.println("additems(2,2,2) = " + calc.additems(2,2,2)); }
}
```

Here's the result of this program:

```
C:\>java App
additems(2,2) = 4
additems(2,2,2) = 6
```

As you can see, overloading provides a powerful technique, especially in code you release to other developers, because being able to pass all different types of parameter lists to a method makes that method easy to use in many ways in code. You can also overload constructors—see the next topic for details.

Overloading Constructors

"Alright," says the Novice Programmer, "so I can overload methods in Java to let them handle different parameter lists! Can I also overload constructors?" "Of course," you say. "Consider the Java `String` class, which has a constructor to which you can pass strings, character arrays, and all other kinds of data." "Oh yeah," says the NP.

Overloading constructors work like overloading other methods (see the previous topic for details): You just define the constructor a number of times, each time with a parameter list with parameters that differ from the other lists in some way.

Here's an example that mimics the Java `String` class's constructors in this new class, the `Data` class, will have a constructor to which you can pass either a character array or a string. This class will simply store the text data you pass to it and make that data available with a `getData()` method.

Here's how we declare and define the constructor that takes a character array:

```
class Data {
  private String data_string;
  Data(char[] c) { data_string = new String(c); }
}
```

Here's how we declare the constructor that takes a text string:

```
class Data {
  private String data_string;
  Data(char[] c) { data_string = new String(c); }
  Data(String s) { data_string = s; }
}
```

All that's left is to add the `getData()` method:

```
class Data {
  private String data_string;
  Data(char[] c) { data_string = new String(c); }
  Data(String s) { data_string = s; }
  public String getData() { return data_string; }
}
```

Now we can use both constructors in code, creating objects and printing out the stored text, like this:

```
public class App {
  public static void main(String[] args) {
        char chararray[] = {'H', 'e', 'l', 'l', 'o'};
        System.out.println((new Data(chararray)).getData());
        System.out.println((new Data("Hello from Java!")).getData()); }
}
```

Here's the result of this code:

```
C:\>java App
Hello
Hello from Java!
```

Passing Objects to Methods

The Novice Programmer appears and says, "Java's gone all wacky again. I passed an object to a method because I want to do a lot of destructive testing on it in that method; but when control returned from the method, the *original* object was destroyed. What happened?" "Java passes objects by reference," you say. "That's what happened."

When you pass an item of a simple data type to a method, Java passes a copy of the data in the item, which is called *passing by value*. Because the method gets only a copy of the data item, the code in the method cannot affect the original data item at all.

However, when you pass an object to a method, Java actually passes a reference to the object, which is called *passing by reference*. Passing by reference means that the code in the method can reach the original object. In fact, any changes made to the passed object affect the original object.

Here's an example in which we pass an object of class `Data` to the `print()` method of the `Printer` class in order to print out the data in the object:

```
class Data {
  public String data_string;
        Data(String data) { data_string = data; }
}
class Printer {
  public void print (Data d) { System.out.println(d.data_string); }
}
public class App {
  public static void main(String[] args) {
        Data data = new Data("Hello from Java!");
        Printer p = new Printer();
        p.print(data);  }
}
```

Here's the result of this code:

```
C:\>java App
Hello from Java!
```

As mentioned previously, because objects are passed by reference, changing a passed object changes the original object. Here's an example in which we pass an object of the `Data` class to a method named `rewrite()` that changes the `data_string` instance variable in the object. This variable starts out with the string "Hello from Java!" in it, but the `rewrite()` method is able to change the string to "Hello to Java!" in this code:

```
class Data {
  public String data_string;
  Data(String s) { data_string = new String(s); }
}
class Data2 {
  public void rewrite(Data d) { d.data_string = "Hello to Java!"; } }
public class App {
  public static void main(String[] args) {
        Data d = new Data("Hello from Java!");
        Data2 d2 = new Data2();
        d2.rewrite(d);
        System.out.println(d.data_string); }
}
```

Here's the result of this code:

```
C:\>java App
Hello to Java!
```

Passing Arrays to Methods

"So, simple variables are passed by value in Java and objects are passed by reference. I have it all straight now," says the Novice Programmer. "Not quite," you say. "There's one more type of item that's passed by reference—arrays."

You can pass an array to a method as easily as passing a simple variable, but you should keep in mind that arrays are passed by reference, not value, which means that if you make a change to an array passed to a method, the original array is also affected.

Here's an example. In this case, we'll pass an array to a method named `doubleValue()` that doubles each element in the array. Because arrays are passed by reference, the data in the original array is doubled as well. Here's how this looks in code (note that we print out an array before and after the call to the `doubleValue()` method):

```
class Calculate {
  public void doubleValue(int a[]) {
        for (int loop_index = 0; loop_index < a.length; loop_index++) {
              a[loop_index] *= 2; }
  }
```

```
}
public class App {
  public static void main(String[] args) {
        int array[] = {5,3,6,1,7};
        Calculate c = new Calculate();
        System.out.println("Before calling the doubleValue()...");
        for (int loop_index = 0; loop_index < array.length; loop_index++) {
              System.out.println("array[" + loop_index + "] = " +
              array[loop_index]); }
        c.doubleValue(array);
        System.out.println("After calling the doubleValue()...");
        for (int loop_index = 0; loop_index < array.length; loop_index++) {
              System.out.println("array[" + loop_index + "] = " +
              array[loop_index]); }
  }
}
```

Here's the result of this code:

```
C:\>java App
Before calling the doubleValue()...
array[0] = 5
array[1] = 3
array[2] = 6
array[3] = 1
array[4] = 7
After calling the doubleValue()...
array[0] = 10
array[1] = 6
array[2] = 12
array[3] = 2
array[4] = 14
```

Using the **this** Keyword

Java objects include a data member named `this` which is actually a reference to the current object. The `this` keyword is useful if you need to refer to the current object—for example, when you want to pass the current object to a method. Here's an example. In this case, the `Data` class has a method, `printData()`, that prints the data in the current object by passing the current object to the `print()` method of another object. The `this` keyword is used to refer to the current object. Here's how this looks in code:

```
class Data {
  private String data_string;
  Data(String s) { data_string = s; }
  public String getData() { return data_string; }
  public void printData() {
        Printer p = new Printer();
        p.print(this);  }
}
class Printer {
  void print(Data d) { System.out.println(d.getData()); }
}
public class App {
  public static void main(String[] args) {
        (new Data("Hello from Java!")).printData(); }
}
```

Note that when the call to `p.print()` is made, a reference to the current object is passed to `p.print()`, giving the code in `p.print()` access to the `getData()` method in the current object, which returns the internal data to print out. Here's the result of this program:

```
C:\>java App
Hello from Java!
```

Returning Objects from Methods

You can return objects from methods, just like other data types. However, this raises a concern: When the method that returned the object goes out of scope, won't the object that it returned also go out of scope? The answer is no. When you create a new object by using the `new` operator, that object is not destroyed when the method that created it goes out of scope, and the object, itself, is not disposed of by the garbage collector until there are no more references to it.

Here's an example. In this case, a class named `ObjectFactory` has a method named `getNewObject()` that returns an object of the class `CreatedClass`:

```
class ObjectFactory {
   public CreatedClass getNewObject() {  return new CreatedClass(); }
}
class CreatedClass { public String tag = "This is the tag data."; }
public class App {
   public static void main(String[] args) {
          ObjectFactory o = new ObjectFactory();
          CreatedClass c = o.getNewObject();
          System.out.println(c.tag); }
}
```

When we call the `getNewObject()` method, it returns a new object of the `CreatedClass` class, and we can print out the data in that object. Here's what is shown when the program runs:

```
C:\>java App
This is the tag data.
```

Returning Arrays from Methods

You can return arrays from methods just as you can return objects and simple data types. Here's an example in which we create a class called `ArrayFactory` with a method named `getNewArray()`. When we call `getNewArray()`, it will return an array of integers. Note the return type we specify for `getNewArray()` in the declaration: `int[]`. This indicates an integer array. Here's the code:

```
class ArrayFactory {
   public int[] getNewArray() {
          int array[] = {11, 12, 13, 14, 15};
          return array; }
}
```

Here's how we put the `ArrayFactory` class to work, creating a new array and printing it out:

```
public class App {
   public static void main(String[] args) {
          ArrayFactory af = new ArrayFactory();
          int array[] = af.getNewArray();
          for (int loop_index = 0; loop_index < array.length; loop_index++) {
                 System.out.println("array[" + loop_index + "] = " +
                 array[loop_index]); }
   }
}
```

Here's the result of this code:

```
C:\>java App
array[0] = 11
array[1] = 12
array[2] = 13
array[3] = 14
array[4] = 15
```

The ProcessBuilder and StringBuilder Classes

"Fine, after going through all these facts there is one query, what are the `ProcessBuilder` and `StringBuilder` classes for?" the Novice Programmer asks. You say, "Ok my dear, let's study both of them.".

Java was enriched the `lang` package by two new classes: "`the ProcessBuilder`" class and "`the StringBuilder`" class.

The *ProcessBuilder* Class

An operating system process is created by using this class. A collection of process attributes is managed by every `ProcessBuilder` instance. By using those attributes, the `start()` method creates instance of a new process. For creating new subprocesses with similar/associated attributes, the `start()` method is invoked in a repeated manner from the same instance.

These process attributes are managed by each process builder in the following ways:

- **Command**—Command consists of a sequence of strings that enable the invocation of external program file and its arguments, if exists. The string lists representing a legitimate operating system command are system-dependent. For instance, generally each conceptual argument is an element in this list, but we can find instances of operating systems where programs are likely to tokenize command-line strings themselves—in such a scenario, a Java implementation might need commands for containing only two elements.
- **Environment**—A mapping, which is system-dependent varying from variables to values. The copy of the environment of current process is the initial value.
- **Working directory**—The current working directory of the current process is the default value; usually, the system property, user.dir, imparts name to this directory.
- `RedirectErrorStream` **property**—At initial stage, standard output and error output of a subprocess are sent to two separate streams, meaning that this property is false. The false property can be accessed by using two of the methods—the `Process.getInputStream()` method and `Process.getErrorStream()` method. The standard error is intermingled with the standard output, if the value is set to true. Thus, the correlation between error messages and the corresponding output is made easier. In this case, the intermingled data can be read by `Process.getInputStream()` which is a returned stream, while we'll get an immediate result of end of file, if we read from the `Process.getErrorStream()` returned stream.

The processes further started by using the `start()` method of object will be affected if attributes of process builder are modified, but processes previously started or the Java process itself will never be affected.

The `start()` method performs most of the error checking. It is very much possible to make the `start()` fail if we change the object's state. For example, if the command attribute is set to an empty list, an exception is not thrown by it unless the `start() method` is invoked.

NOTE

This class is not at all synchronized. If an instance of the ProcessBuilder class is retrieved concurrently by multiple threads and at least one of the attributes is modified structurally by one of the threads, it must be externally synchronized.

It is very easy to start a new process, which make use of both the default working directory and environment. For example:

```
import java.io.*;
public class App {
  public static void main(String args[])
  throws IOException {
        ProcessBuilder pb=new ProcessBuilder("notepad.exe", "nfile.java");
        pb.start(); }
}
```

In the previous program, we have used an exception statement which you'll study later in detail, but for information we're giving you a brief idea. An exception refers to an object that is created whenever an error occurs in a Java program and cannot be fixed automatically. An exception object contains the information about the error. Exceptions that occur belong to different classes according to the occurrence of errors. A `throws` keyword used here simply lists the exceptions that the method might throw. The `throws` statement has the following basic format:

```
throw new exception-class();
```

When the program is executed, it will create a new file or open that if it already exists within the same directory. You'll get a dialogue box as shown in Figure 4.1.

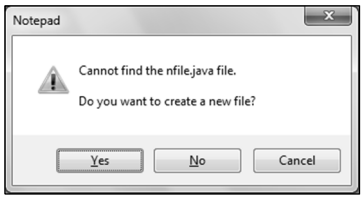


Figure 4.1: Dialog box prompts to create a file

The *StringBuilder* Class

The thread-safe `StringBuffer` class has a replacement in the form of `StringBuilder`. It does not contain synchronized methods, so it works much faster. So, if you are performing a number of `String` operations in a single thread, you'll gain a tremendous performance while using this class.

The `StringBuilder` class consists of mutable sequence of characters. It provides an API, which is compatible with `StringBuffer`, but there is no guarantee of its synchronization. This class is designed in such a way that it can be used as a drop-in replacement for `StringBuffer` in places where a single thread is using the `StringBuffer` (which is generally the case). It is recommended that this class get precedence over `StringBuffer`, as under most implementations, it will be faster.

The `append()` and `insert()` methods are the principal operations in a `StringBuilder`, which are overloaded so that these can accept data of any type. A given datum is effectively converted into a string by each of these methods and then the characters of that string are appended or inserted to the `StringBuilder class`. These characters are always added at the end of the builder by the `append()` method, whereas the characters are added at a specified point by the `insert()` method.

For example, if we consider `z` as a `StringBuilder` object whose "start" are current contents, then the method that cause the `StringBuilder` to contain "startle" is called `z.append("le")`, whereas the method that alter the `StringBuilder` to contain "starlet" is called `z.insert(4, "le")`.

In general, if an instance of a `StringBuilder` class is referred to by `sbd`, then sbd.append(z) behaves in the same way as `sbd.insert(sbd.length(), z)`. There is a capacity in every `StringBuilder`. It is not mandatory to assign a new internal buffer until the length of the character sequence present in the `StringBuilder` class does not surpass the capacity. The `StringBuilder` gets larger automatically, if there is an overflow of internal buffer.

It is not advised to use the instances of `StringBuilder` by multiple threads. It is recommended that `StringBuffer` be used in case such synchronization is needed.

NOTE

You can create new objects of this class from any of the four classes that implements the interface, as "the StringBuilder class" has constructors with a parameter of type CharSequence.

Here's an example showing the concatenation of a list of strings into a single string, when string is declared by using `StringBuilder`. The following code demonstrates this use, to give you more exposure to `StringBuilder`:

```
class App {
  public static void main (String [] args) {
        System.out.println ("Concatenation: " +
        concat ("Hello", " ", "Java")); }
  static String concat (String ... strings) {
        StringBuilder sb = new StringBuilder ();
  for (int i = 0; i < strings.length; i++)
              sb.append (strings [i]);
        return sb.toString (); }
}
```

Here's the result of this code:

```
C:\>java App
Concatenation: Hello Java
```

Let's take one more program of the `StringBuilder` class:

```
import java.util.Scanner;
public class App {
  static Scanner scan = new Scanner(System.in);
  public static void main(String[] args) {
        System.out.print("Enter a string: ");
        String str = scan.nextLine();
        StringBuilder sb = new StringBuilder(str);
                        for (int index_of_string = 0; index_of_string
        < str.length(); index_of_string++) {
                char chr = str.charAt(index_of_string);
                if ( (chr == 'A') || (chr == 'a')
                        || (chr == 'E') || (chr == 'e')
                        || (chr == 'I') || (chr == 'i')
                        || (chr == 'O') || (chr == 'o')
                        || (chr == 'U') || (chr == 'u') ) {
                                sb.setCharAt(index_of_string, '*'); }
  }
  System.out.println();
  System.out.println(str);
  System.out.println(sb.toString()); }
}
```

In the preceding code, we have used a special class called the `Scanner` class; let's give you a brief introduction of it.

You can read input from either the keyboard or a text file by using `Scanner` class. This class provides a simple and standard mechanism for reading input; the complete `Scanner` class can be accessed by importing `java.util.Scanner`. The input received by the `Scanner` class is broken into tokens. For breaking, `Scanner` uses delimiter pattern, which by default matches whitespace. These tokens are further converted into values of various types by using the following methods.

The following code allows a user to read a number from `System.in`:

```
Scanner scan = new Scanner(System.in);
int i = scan.nextInt();
```

When the program is compiled and run, it involves an interactive session. So, on entering any string, you'll get the symbol '`*`', at the ('`a`','`e`','`i`','`o`','`u`') vowel's position. The output is shown here:

```
C:\>java App
Enter a string: Kogent
Kogent
K*g*nt
```

Catching an Exception

"No!," says the Novice Programmer. "My program just crashes when I divide by zero." "Well," you say, "it's pretty hard for Java to know what to do when you do that. Why not catch `ArithmeticException` exceptions and handle divisions by zero yourself?" The NP says, "Tell me more!"

You can use `try/catch` blocks to handle runtime errors, called *exceptions*. Exceptions are encapsulated into objects that extend the `Throwable` class. In fact, there's another kind of runtime error in Java that's built on the `Throwable` class—it's called `Error`. However, this type of error is serious and cannot be caught in code. Here's the inheritance diagram for the `Throwable` class:

```
java.lang.Object
  |____java.lang.Throwable
```

You'll find the constructors of the `Throwable` class in Table 4.2 and its methods in Table 4.3. Note, in particular, the `getMessage()` method, which returns an error message you can display to the user, and the `toString()` method, which lets you print out an exception object as part of a string, like this:

```
System.out.println("Exception: " + e)
```

Here, `e` is an exception object.

Table 4.2: Constructors of the Throwable class

Constructor	Does this
`Throwable()`	It constructs a new `Throwable` object
`Throwable(String message)`	It constructs a new `Throwable` object with the indicated error message
`Throwable(String message , Throwable cause)`	It constructs a new `Throwable` object with the specified detail message and cause
`Throwable(Throwable cause)`	It constructs a new `Throwable` object with the specified cause and detail `messageof(cause==null ?null:cause.toString())` (which typically contains the class and detail message of cause)
`protected Throwable (String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace)`	It constructs a new `Throwable` object with the indicated error message, cause, suppression disabled, or enabled, and writable stack trace disabled or enabled

Table 4.3: Methods of the Throwable class

Method	Does this
`void addSuppressed(Throwable exception)`	It appends the indicated exception to the exceptions that were suppressed to deliver this exception
`Throwable fillInStackTrace()`	It fills in the execution stack trace
`Throwable getCause()`	It yields the cause of this `Throwable` object or null in case the cause doesn't exist or not known
`String getLocalizedMessage()`	It gets a localized description of this `Throwable` object
`String getMessage()`	It gets the error message string of this `Throwable` object
`StackTraceElement[] getStackTrace()`	It provides programmatic access to the stack trace information printed by `printStackTrace()`
`Throwable[] getSuppressed()`	It returns an array containing all the exceptions suppressed to deliver this exception
`Throwable initCause (Throwable cause)`	It initializes the cause of this throwable to the specified value
`void printStackTrace()`	It prints the stack trace to the standard error stream
`void printStackTrace (PrintStream s)`	It prints the stack trace to the indicated print stream
`void printStackTrace (PrintWriter s)`	It prints the stack trace to the indicated print writer
`void setStackTrace (StackTraceElement [] stackTrace)`	It sets the stack trace elements that will be returned by `getStackTrace()` and printed by `printStackTrace()` and related methods
`String toString()`	It gets a short description of this `Throwable` object

Here's the inheritance diagram for the `Exception` class:

```
java.lang.Object
  |____java.lang.Throwable
        |____java.lang.Exception
```

You'll find the constructors of the `Exception` class in Table 4.4. Here's how you use a `try/catch` block in general:

```
try { } // Sensitive code
catch (Exception1Type e1) { } // Handle exception1 type
catch (Exception2Type e1) { } // Handle exception2 type
. . .
finally { } // Code to be executed after try-catch block ends
```

Table 4.4: Constructors of the Exception class	
Constructor	**Does this**
`Exception()`	It constructs an `Exception` object with null as its detail message
`Exception(String s)`	It constructs an `Exception` object with the indicated detail message
`Exception(String message, Throwable cause)`	It constructs a new `Exception` object with the specified detail message and cause
`protected Exception(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace)`	It constructs a new `Exception` object with the specified detail message, cause, suppression enabled or disabled, and writable stack trace enabled or disabled
`Exception (Throwable cause)`	It constructs a new exception with the specified cause and a detail message of (cause==null ? null : cause.toString()) (which typically contains the class and detail message of cause)

When some sensitive code inside a `try` block throws an exception, you can catch it with a `catch` block. You can also use a `finally` block to execute code after the `try/ catch` block is complete, and the code in the `finally` block will be executed whether or not an exception occurred. You can use a `finally` block to handle exceptions not explicitly caught in a `catch` block. There are many, many different types of exceptions in Java, and you'll find some of the exception classes in Table 4.5.

Table 4.5: Some Java exception classes		
`AclNotFoundException`	`ActivationException`	`AlreadyBoundException`
`ApplicationException`	`AWTException`	`BackingStoreException`
`BadLocationException`	`CertificateException`	`ClassNotFoundException`
`CloneNotSupportedException`	`DataFormatException`	`DestroyFailedException`
`ExpandVetoException`	`FontFormatException`	`GeneralSecurityException`
`GSSException`	`IllegalAccessException`	`InstantiationException`
`InterruptedException`	`IntrospectionException`	`InvalidMidiDataException`
`InvalidPreferencesFormatException`	`InvocationTargetException`	`IOException`
`LastOwnerException`	`LineUnavailableException`	`MidiUnavailableException`
`MimeTypeParseException`	`NamingException`	`NoninvertibleTransformException`
`NoSuchFieldException`	`NoSuchMethodException`	`NotBoundException`
`NotOwnerException`	`ParseException`	`ParserConfigurationException`
`PrinterException`	`PrintException`	`PrivilegedActionException`
`PropertyVetoException`	`RefreshFailedException`	`RemarshalException`
`RuntimeException`	`SAXException`	`ServerNotActiveException`
`SQLException`	`TooManyListenersException`	`TransformerException`
`UnsupportedAudioFileException`	`UnsupportedCallbackException`	`UnsupportedFlavorException`
`UnsupportedLookAndFeelException`	`URISyntaxException`	`UserException`
`XAException`		

Take a look at this simple example that divides two numbers and uses `try/catch` statement to catch an exception, if the second number turns out to be zero:

```
public class Divide {
  public static void main(String[] args) {
        int d = 5;
        int z = 0;
```

```
                try { int quotient = d/z; }
            catch (ArithmeticException e) { System.out.println ("Oops!!"); }
    }
}
```

Here's the output of the preceding code:

```
C:\> java Divide
Oops!!
```

Another example that uses a method to get a valid integer from the user is as follows:

```
import java.util.*;
public class VerifyInt {
   static Scanner scan = new Scanner(System.in);
   public static void main(String[] args) {
          System.out.print("Enter an integer: ");
          int x = VerifyInt();
          System.out.println("You have entered " + x); }
   public static int VerifyInt() {
          while (true) {
                        try { return scan.nextInt(); }
                catch (InputMismatchException e) {
                        scan.next();
                        System.out.print("It is not an integer. "
                        + "Try again: "); } }
   }
   }
```

Here's the result:

```
C:\> java VerifyInt
Enter an integer: 2.0
It is not an integer. Try again: 14.0
It is not an integer. Try again: 7.6
It is not an integer. Try again: 20
You have entered 20
```

In this case, if the user enters a non-integer value, the `catch` block catches the error and forces the loop to repeat. The `nextInt()` method throws an `InputMismatchException`, if the user has entered data that cannot be converted into an integer, . Note that the program imports all the classes from the `java.util` package, including the `MismatchException` class. When an `InputMismatchException` is thrown, the `nextInt()` method leaves the input value in the `Scanner's` input stream. Then the `scan.next()` method in the `catch` block disposes of the user's invalid input. If this method (`scan.next()`) is omitted, the while loop keeps reading the input stream, throws an exception, and displays an error message in an infinite loop.

Here's one example of catching an exception. In this case, we're catching an "array index out of bounds" exception:

```
public class Excep {
   public static void main(String args[]) {
          try {
                  int array[] = new int[100];
                  array[100] = 100; }
          catch (ArrayIndexOutOfBoundsException e) {
                  System.out.println("Exception: " + e.getMessage());
                  e.printStackTrace(); }
   }
}
```

When you run this example, here's the result you get, which includes an error message and a stack trace that indicates where the exception occurred:

```
C:\>java Excep
Exception: 100
java.lang.ArrayIndexOutOfBoundsException: 100
        at Excep.main(Excep.java:5)
```

You can also pass exceptions back to methods that called the current method with the `throws` keyword. Here's an example in which we indicate that the `doSomeWork()` method can throw an exception of class `ArithmeticException`, which will be caught in the calling method (or, if not, by the default exception handler

in Java), by specifying the `throws` keyword in the method's definition (note that if you use the `throws` keyword like this, you don't need a `try/catch` block in the method's body):

```
public class Excep2 {
  public static void main(String args[]) {
        try { doSomeWork(); }
        catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Exception: " + e.getMessage());
                e.printStackTrace(); } }
  static void doSomeWork() throws ArithmeticException {
        int array[] = new int[100];
        array[100] = 100; }
}
```

Here's the result of this code. Note that the stack trace indicates that the exception occurred in the `doSomeWork()` method:

```
C:\>java Excep2
Exception: 100
java.lang.ArrayIndexOutOfBoundsException: 100
     at Excep2.doSomeWork(Excep2.java:9)
     at Excep2.main(Excep2.java:3)
```

Here is another example by using the `FileNotFoundException`. A string is passed to the constructor that contains the path and name of a file that exists on your computer. In case the file is not found, a `FileNotFoundException` is thrown. You'll get acquainted with file operations later:

```
import java.io.*;
public class FileExcep {
  public static void main(String[] args) { openFile("test.bmp"); }
  public static void openFile(String name) {
        FileInputStream f = new FileInputStream(name); }
}
```

You'll notice that you cannot compile this example successfully, as the compiler displays the following error:

```
C:\> javac FileExcep.java
FileExcep.java:5: error: unreported exception FileNotFoundException; must be caught or
  declared to be thrown
             FileInputStream f = new FileInputStream(name); }
                                 ^
1 error
```

This message means that you need to take care of the `FileNotFoundException`. To do so, you need to catch this exception by using the `try` statement:

```
import java.io.*;
public class FileExcep {
  public static void main(String[] args) { openFile("test1.bmp"); }
  public static void openFile(String name) {
        try { FileInputStream f = new FileInputStream(name); }
        catch (FileNotFoundException e) {
                System.out.println("File not found."); }
  }
}
```

On executing this example, the following result gets displayed:

```
C:\>java FileExcep
File not found.
```

Nesting the **try** Statements

You can have nested `try` blocks inside other `try` blocks. If one `try` block doesn't have a corresponding `catch` block that handles an exception, Java searches the next outer `try` block for a `catch` block that will handle the exception (and so on, back through the successive nestings). If Java can't find a `catch` block for the exception, it will pass the exception to its default exception handler. Here's an example of nested `try` blocks:

```
public class NestedExcep {
  public static void main(String args[]) {
        try {
                try { int num[] = {10, 12, 24, 31}; num[4] = 47; }
```

```
                    catch(ArrayIndexOutOfBoundsException e) {
                            System.out.println("Array index out of bounds: "
                            + e); } }
            catch(ArithmeticException e) {
                    System.out.println("Divide by zero: " + e); }
      }
    }
```

Here's the result of this code:

```
C:\>java NestedExcep
Array index out of bounds: java.lang.ArrayIndexOutOfBoundsException: 4
```

Using the **finally** Clause

The `finally` block comes after all the `catch` blocks. Irrespective of whether any exceptions are thrown by the `try` blocks or caught by the `catch` blocks, this statement is executed. The main aim is to wrap up the tasks that are not completed by the exception, such as opened files or database connections. Here's an example using the `finally` clause:

```
import java.util.*;
public class FinExcep {
   static Scanner scan = new Scanner(System.in);
   public static void main(String[] args) {
                        try {
                         System.out.print("\nEnter an integer: ");
                         int i = getInt();
                         System.out.print("Enter another integer: ");
                         int j = getInt();
                         divideTheseNumbers(i,j); }
                        catch (Exception e) {
                         System.out.println("...still it didn't work!"); }
   }
   public static int getInt() {
                        while (true)
                        try { return scan.nextInt(); }
                        catch (InputMismatchException e) { scan.next(); }
   }
   public static int divideTheseNumbers(int a, int b) throws Exception {
                        int c;
                        try { c = a / b; System.out.println("\n...It worked!!"); }
                        catch (Exception e) {
                         System.out.println("\nOops!! Didn't work!");
                         c = a / b;
                         // System.out.println("Oops!! Didn't work again!");
                            }
                        finally { System.out.println("\n...Given my best try....."); }
                        System.out.println("It worked after all.");
                        return c; }
}
```

Here's the result of this code:

```
C:\> java FinExcep
Enter an integer: 12
Enter another integer: 2

...It worked!!

...Given my best try.....
It worked after all.

C:\> java FinExcep
Enter an integer: 9
```

```
Enter another integer: 0

Oops!! Didn't work!

...Given my best try.....
...still it didn't work!
```

Throwing Exceptions

"Argh! That Johnson is at it again," says the Novice Programmer, "abusing my code." "Well," you say, "you can throw exceptions yourself in the code if you want to." "Yes?" asks the NP, "Tell me more!"

You can throw your own exceptions with the `throw` statement. Here is an example in which we throw an exception that will be caught in a `catch` block that, in turn, throws the exception again to be caught in the `calling` method:

```
public class ThrowExcep {
  public static void main(String args[]) {
        try { doSomeWork(); }
        catch(ArithmeticException e) {
                System.out.println("Exception caught in main" + e); } }
  static void doSomeWork() {
        try { throw new ArithmeticException("Arithmetic Exception!"); }
        catch(ArithmeticException e) {
                System.out.println("Caught inside doSomeWork " + e);
                throw e; } }
}
```

Here's the output of this code:

```
C:\>java ThrowExcep
Caught inside doSomeWork java.lang.ArithmeticException: Arithmetic Exception!
Exception caught in mainjava.lang.ArithmeticException: Arithmetic Exception!
```

Let's now throw the `FileNotFoundException` that we dealt with earlier in this chapter. If we pass this exception to the `main()` method instead of the `openFile()` method, the example would be as follows:

```
import java.io.*;
public class FileExcep1 {
  public static void main(String[] args) {
        try { openFile("test1.bmp"); }
        catch (FileNotFoundException e) {
                System.out.println("File not found."); } }
  public static void openFile(String name) throws FileNotFoundException {
        FileInputStream f = new FileInputStream(name); }
}
```

Adding a throws clause to the `openFile()` method means that when the `FileNotFoundException` occurs, it is simply passed to the method that called the `openFile()` method. That means the `calling` method must either catch or throw the exception. In case the file does not exist, the `catch` block catches the exception and displays the error message, as shown in the result:

```
C:\> java FileExcep1
File not found.
```

In fact, the throws clause lists all the exception(s) that the method might throw. In case more than one exception is on the list, it can be separated by commas like this:

```
public static void readFile(String name) throws
FileNotFoundException, IOException
```

You can throw a `FileNotFoundException` from the `main` method also, by using a throws clause like this:

```
public static void main (String[] args) throws FileNotFoundException {
  openFile("test1.bmp");
}
```

"Can we ignore an exception?" asks NP. "Of course you can!" you say. NP looks satisfied.

You can ignore an exception by catching the exception in the `catch` block of a `try` statement and then leaving the `catch` block empty:

```
public static void ope1(String name) {
   try { FileInputStream f = new FileInputStream(name); }
   catch (FileNotFoundException e) { }
}
```

Note that you don't have to throw any of the predefined Java exceptions—you can define your own exceptions. See the next solution for the details.

Creating a Custom Exception

"See," says the NP, "Java just doesn't have the exception I need to use." "What is it?" you ask. "The temperature-too-high exception," the NP says. "Hmm," you say, "how thoughtless of Java not to include that. However, you can create your own exception classes, so you're OK." "Great!" says the NP.

You can create your own exception classes by extending the `Exception` class— just provide a constructor and override the methods you plan to use, such as `getMessage()` and `toString()`.

Here's an example:

```
class Excep extends Exception  {
   int value;
   Excep(int val) {value = val; }
   public String toString() { return "Newly generated exception " + value; }
}
public class CustomExcep {
   public static void main(String args[]) {
         try {
                doSomeWork(3);
                doSomeWork(2);
                doSomeWork(1);
                doSomeWork(0);
            }
                catch (Excep e) { System.out.println("Exception: " + e); } }
                static void doSomeWork(int value) throws Excep {
                if(value == 0) { throw new Excep(value); }
                       else { System.out.println("No problem."); } }
}
```

Here's the output of this example:

```
C:\>java CustomExcep
No problem.
No problem.
No problem.
Exception: Newly generated exception 0
```

Catching Multiple Exception Types

"Can a single `catch` block handle more than one kind of exception?" asks the NP. "Till Java SE 6, this facility was not available in Java, but Java SE 7 and Java SE 8 have this facility," you say. "That's good," says the NP.

The "multicatch" is an exciting new feature in Java SE 7. This Java enhancement lets you catch different types of exception using a single `catch` block. It reduces code duplication and temptation to catch overly broad exceptions, such as `catch(Exception e)`. Sample code lines that show catching multiple exceptions prior to Java 7 is as follows:

```
catch(SQLException e) {  . . . }
catch(IOException e)  {  . . . }
```

Here's how you can catch multiple exceptions in a single `catch` block:

```
catch(SQLException|IOException e) {  . . . }
```

You can see that in Java SE 7, a single `catch` block allows you to specify different types of exceptions separated by a vertical bar (`|`) symbol. Also, note that the `catch` parameter `e` is implicitly final. Therefore, you can't assign or modify value contained in it inside the `catch` block. The bytecode generated on compilation of the `catch` block handling multiple exception types is smaller in size in comparison to the bytecode generated on compilation of various `catch` blocks that are handling the single exception type. The bytecode generated in case of Java SE 7 `catch` block handling multiple exceptions contains no duplication or replication of exception handlers.

Rethrowing Exceptions with Improved Type Checking

"Can I rethrow an exception?" asks the NP. "Hmm," you say, "you can; in fact, Java SE 7 allows you to rethrow exceptions in a better way." "How?" asks the NP.

In Java SE 7, the compiler analyzes the rethrown exceptions in a much improved manner compared to the previous versions of Java. This newly added "final rethrow" feature allows you to specify more specific exception types in the throws clause. Consider the following example to understand better:

```
class Exception1 extends Exception  {
  Exception1(String message) {  super(message); }
}
class Exception2 extends Exception  {
  Exception2(String message) {  super(message); }
}
```

Suppose there is a method rethrowMyException which is shown as:

```
public void rethrowMyException(String exname) throws Exception {
   try {
     if(exname.equals("Exception1") { throw new Exception1("Exception1"); }
     else  {  throw new Exception1("Exception2"); }
    }
   catch(Exception e) { throw e; }
}
```

Here, the try block throws either Exception1 or Exception2 depending upon the name which it receives as parameter. Before Java SE 7, you couldn't specify the exception types following the rethrowMyException() method declaration; you could only write Exception there in this situation. This is because the exception parameter e in the catch block is of type Exception which the catch block rethrows.

Java SE 7 allows you to specify the Exception1 and Exception2 in the throws clause while declaring method rethrowMyException(). The compiler can then deduce that the exception must have been generated in the try block and is of the type either Exception1 or Exception2 even though the exception parameter e in the catch block is of the type Exception. Therefore, in Java SE 7, the rethrowMyException() method can be written as follows:

```
public void rethrowMyException(String exname) throws Exception1, Exception2 {
   try {
     . . .
    }
   catch(Exception e) { throw e; }
}
```

Note that if you are specifying a different exception type (not Exception) in the catch block, then the throws clause after the declaration of method rethrowMyException() must have exception type Exception. In Java SE 7, when you rethrow an exception, the compiler verifies whether the try block can throw that exception or not; there is not any preceding catch block that can handle that exception, and the exception is a subtype or supertype of the exception type of an exception parameter in any of the catch blocks.

Debugging Java Programs

"Uh-oh," says the NP, "I have got a logic problem in my code." "You mean a *bug*," you say. "I prefer to think of it as a logic problem," says the NP. "OK," you say. "Either way, it's time to use the Java debugger."

The Java debugging tool is `jdb`, and you can use it much like you use the `java` tool, except `jdb` lets you single step through your code, install breakpoints to stop execution at a particular line, and more. Here's an example in which we use the debugger on an application:

```
public class App {
  public static void main(String[] args) {
        int a = 51, b = 22, c = 34, d = 45;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d); }
}
```

To start with, we compile the application with the `-g` option, which makes Java include debugging information (it's not necessary to use `-g`, but the debugger will often suggest it if you don't use it):

```
C:\>javac -g -deprecation App.java
```

Now we start `jdb` just as we would start the `java` tool. One of the best ways to see what's available in the Java debugger is to ask it to explain itself, which we do with the `help` command at the `jdb` prompt (see the highlighted line of code):

```
C:\>jdb App
Initializing jdb ...
> help
** command list **
connectors                -- list available connectors and transports in this VM


run [class [args]]        -- start execution of application's main class

threads [threadgroup]     -- list threads
thread <thread id>        -- set default thread
suspend [thread id(s)]    -- suspend threads (default: all)
resume [thread id(s)]     -- resume threads (default: all)
where [<thread id> | all] -- dump a thread's stack
wherei [<thread id> | all]-- dump a thread's stack, with pc info
up [n frames]             -- move up a thread's stack
down [n frames]           -- move down a thread's stack
kill <thread id> <expr>   -- kill a thread with the given exception object
interrupt <thread id>     -- interrupt a thread
print <expr>              -- print value of expression
dump <expr>               -- print all object information
eval <expr>               -- evaluate expression (same as print)
set <lvalue> = <expr>     -- assign new value to field/variable/array element
locals                    -- print all local variables in current stack frame

classes                   -- list currently known classes
class <class id>          -- show details of named class
methods <class id>        -- list a class's methods
fields <class id>         -- list a class's fields

threadgroups              -- list threadgroups
threadgroup <name>        -- set current threadgroup

stop in <class id>.<method>[(argument_type,...)]
                          -- set a breakpoint in a method
```

```
stop at <class id>:<line> -- set a breakpoint at a line
clear <class id>.<method>[(argument_type,...)]
                          -- clear a breakpoint in a method
clear <class id>:<line>   -- clear a breakpoint at a line
clear                     -- list breakpoints
catch [uncaught|caught|all] <class id>|<class pattern>
                          -- break when specified exception occurs
ignore [uncaught|caught|all] <class id>|<class pattern>
                          -- cancel 'catch' for the specified exception
watch [access|all] <class id>.<field name>
                          -- watch access/modifications to a field
unwatch [access|all] <class id>.<field name>
                          -- discontinue watching access/modifications
                         to a field
trace [go] methods [thread]
                          -- trace method entries and exits.
                          -- All threads are suspended unless 'go' is specified
                         trace [go] method exit | exits [thread]
                          -- trace the current method's exit, or all methods'
                           exits
                          -- All threads are suspended unless 'go' is specified
untrace [methods]         -- stop tracing method entrys and/or exits
step                      -- execute current line
step up                   -- execute until the current method returns to
 its caller
stepi                     -- execute current instruction
next                      -- step one line (step OVER calls)
cont                      -- continue execution from breakpoint
list [line number|method] -- print source code
use (or sourcepath) [source file path]
                          -- display or change the source path
exclude [<class pattern>, ... | "none"]
                          -- do not report step or method events for specified
                         classes
classpath                 -- print classpath info from target VM

monitor <command>         -- execute command each time the program stops
monitor                   -- list monitors
unmonitor <monitor#>      -- delete a monitor
read <filename>           -- read and execute a command file

lock <expr>               -- print lock info for an object
threadlocks [thread id]   -- print lock info for a thread

pop                       -- pop the stack through and including the current
                         frame
reenter                   -- same as pop, but current frame is reentered
redefine <class id> <class file name>
                          -- redefine the code for a class

disablegc <expr>          -- prevent garbage collection of an object
enablegc <expr>           -- permit garbage collection of an object

!!                        -- repeat last command
<n> <command>             -- repeat command n times
# <command>               -- discard (no-op)
help (or ?)               -- list commands
```

```
version                    -- print version information
exit (or quit)             -- exit debugger
<class id>: a full class name with package qualifiers
<class pattern>: a class name with a leading or trailing wildcard ('*')
<thread id>: thread number as reported in the 'threads' command
<expr>: a Java(TM) Programming Language expression.
Most common syntax is supported.
Startup commands can be placed in either "jdb.ini" or ".jdbrc"
in user.home or user.dir
```

Then set a breakpoint in the `main` method like this:

```
> stop in App.main
Deferring breakpoint App.main.
It will be set after the class is loaded.
```

Now we can run the code with the `run` command, which will run and stop immediately because of the breakpoint, like this:

```
> run
run App
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint App.main

Breakpoint hit: "thread=main", App.main(), line=3 bci=0
3           int a = 51, b = 22, c = 34, d = 45;

main[1]
```

Now we can use the `next` command to single-step through the code like this:

```
main[1] next
>
Step completed: "thread=main", App.main(), line=4 bci=13
4           System.out.println("a = " + a);

main[1] next
> a = 51

Step completed: "thread=main", App.main(), line=5 bci=38
5           System.out.println("b = " + b);
main[1] next
> b = 22

Step completed: "thread=main", App.main(), line=6 bci=63
6           System.out.println("c = " + c);

main[1] next
> c = 34

Step completed: "thread=main", App.main(), line=7 bci=88
7          System.out.println("d = " + d); }

main[1]
```

As you can see, there are all kinds of debugging options. Try it yourself. The best way to learn how to use `jdb` is to put it to work.

Summary

In this chapter, you have been introduced to objects, data members, methods, inheritance, exception handling, and debugging. You have learned how to declare and define a class, create instance variables, implement variable scope, create class variable, and perform basic operations on methods. We have also discussed about the `this` keyword and operating methods using objects and arrays. You have learned how to perform basic operations on constructors and use garbage collection, `finalize()` method and memory management. Towards the end, the new classes in the `java.lang` package and exception handling in Java has been explored.

The next chapter will discuss about the inheritance, inner classes, and interfaces.



5

Inheritance, Inner Classes, and Interfaces

If you need an immediate solution to:	*See page:*
Creating a Subclass	150
Access Specifiers and Inheritance	150
Calling Superclass Constructors	151
Creating Multilevel Inheritance	153
Handling Multilevel Constructors	154
Overriding Methods	155
Accessing Overridden Members	155
Using Superclass Variables with Subclassed Objects	156
Dynamic Method Dispatch (Runtime Polymorphism)	157
Creating Abstract Classes	158
Stopping Overriding with final	159
Stopping Inheritance with final	159
Creating Constants with final	160
Is-a vs. Has-a Relationships	160
The Java Object Class	161
Using Interfaces for Multiple Inheritance	162
Creating Iterable Objects	166
Creating Inner Classes	168
Creating Anonymous Inner Classes	168

In Depth

This chapter is all about inheritance, a very important topic in Java programming. Inheritance is used to derive one class from another class. The class from which another class is being derived is called the *base class* or *superclass*, and the class that is derived is called the *derived class* or *subclass*. The idea here is that you add what you want to the new class to give it more customized functionality in addition to the original class.

The previous chapter began our discussion of Object-Oriented Programming (OOP) and as we mentioned there, if you have a class named, say, `Vehicle`, that contains the basic functionality of some means of transport, you can use that class as the base class of classes you derive from that class, such as `Car` and `Truck`. The `Car` class might, for instance, have a data member named `wheels`, set to 4, whereas the same data member in the `Truck` class might be set to 18. You can also use the same `Vehicle` class as the base class for other classes, such as `Helicopter` class. All the subclasses will have access to the non-private members of the superclass, and they can add their own. In fact, they can *override* the non-private methods of the superclass, replacing them with their own code. For example, the `Vehicle` class may have a method named `go` that prints out "Driving" and the `Helicopter` class may override that method, redefining it so it prints out "Flying".

Using inheritance, then, you can base your classes on other classes, reusing code and adding to it. You can use or redefine the members of the superclass, as you like, customizing that class for your own use. In fact, you can create classes that must be treated as superclasses. These classes are called *abstract classes*. You can't instantiate an abstract class directly into an object; you must instead derive a new class from it first, overriding those members that are specifically declared abstract. You use abstract classes to force developers to customize some or all of the members of a class; for example, you may have an abstract method named `printError()` because you want developers to supply their own code for this method as appropriate for the subclasses they create.

That's an overview of what inheritance does. The next question arises, why is inheritance so important in Java?

Why Inheritance?

Java, being an OOP language, relies heavily on inheritance. It contains a large number of packages which contain classes that you can use as superclasses.

This is important if, for example, you want to create an applet in Java, because in that case, you can derive your applet from the `java.applet` package's `Applet` class. Here's an applet that creates a superclass on the basis of the `Applet` class by using the `extends` keyword (more on applets in the next chapter):

```
import java.applet.Applet;
import java.awt.*;
public class App extends Applet {
        public void paint(Graphics g) {
  g.drawString("Hello Java Reader!", 40, 100); }
}
```

Here's another example; in this case, we are creating a windowed application and basing the window itself on the `Java java.awt.Frame` class:

```
import java.awt.*;
import java.awt.event.*;
class AppFrame extends Frame {
        public void paint(Graphics g) {
  g.drawString("Hello Java Reader!", 40, 100); }
}
public class App {
        public static void main(String [] args) {
  AppFrame f = new AppFrame();
  f.setSize(200, 200);
  f.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
        System.exit(0); }});
  f. setVisible(true); }
}
```

As you can see, when it comes to visual elements in your programs, you'll rely on the Java packages a great deal. Buttons, for example, have their own classes and to customize them, you can derive your own classes from them. In fact, if you even want to handle mouse actions or button clicks, you have to use inheritance — this time, not using superclasses but rather *interfaces*.

Why Interfaces?

Suppose you want to create an applet that handles button clicks. For this, you can create a standard applet by deriving a class from the `java.applet.Applet` class. You can use `ActionListener` class for handling the button clicks. Therefore, both the `Applet` and `ActionListener` classes act as base for your applet.

However, basing a subclass on two or more superclasses is called *multiple inheritance*, and it turns out that Java doesn't support multiple inheritance (although languages, such as C++ do). In practice, this means you can only use the `extends` keyword with one class. To solve this problem, Java implements classes such as `ActionListener` as *interfaces*. In this case, you can extend your applet from the `Applet` class and use the `implements` keyword to add the button-click handling. Here's what this looks like in an applet:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Clicker extends Applet implements ActionListener {
         TextField text1; Button button1;
         public void init() {
  text1 = new TextField(20); add(text1);
  button1 = new Button("Click Here!"); add(button1);
  button1.addActionListener(this); }
         public void actionPerformed(ActionEvent event) {
  String msg = new String ("Welcome to Java");
  if(event.getSource() == button1) { text1.setText(msg); }
  }
}
```

You can implement as many interfaces as you like; for example, here's part of a program that implements three listeners to enable the program to handle button clicks and mouse actions:

```
import java.awt.Graphics;
import java.awt.*;
import java.awt.event.*;
import java.lang.Math;
import java.applet.Applet;
public class dauber extends Applet implements ActionListener,
MouseListener, MouseMotionListener {
   . . .
```

You'll see how to create interfaces later in the book. However, you'll get an introduction to them in this chapter so that we can use them in the chapters to come. There's one more topic we'll cover in this chapter—Inner Classes.

Why Inner Classes?

Java lets you create classes within classes, and the enclosed class is called an *inner class*. We'll start working with inner classes in this chapter. You might not see much need for defining classes within classes now, but it will become more apparent when we start handling user interface events, such as when the user closes a window. Events are handled with interfaces, and when you implement an interface, you must also provide implementations of several abstract methods inside the interface. To make this process easier, Java provides *adapter classes*, which already have empty implementations of the required methods. To handle a user interface event of some kind, it's become common to subclass adapter classes as inner classes, overriding just the methods you want in a very compact way. Here's an example in which the code ends an application when the user closes that application's window:

```
import java.awt.event.*;
public class App {
         public static void main(String [] args) {
  AppFrame f = new AppFrame();
```

```
        f.setSize(200, 200);
        f.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
        System.exit(0); }});
        f.setVisible(true); }
    }
```

We'll unravel this code in detail when working with events, and we'll get started on that process by introducing inner classes so that this code will make much more sense to you later.

Lambda Expressions

A lambda expression refers to a method that has no name; no access specifier private, public, or protected; and no return value declaration. This type of method is also known as 'anonymous method' or 'closure'. Like anonymous class, a lambda expression can be used for performing a task without a name. It is a new and significant feature that is included in Java SE 8. Through lambda expressions, functionality can be passed as method arguments and code as data.

NOTE

An anonymous class does not have a name, but an object of this class can be created.

Consider the following code snippet for understanding the concept of lambda expression, in which a simple method is created for showing a message. Let's first declare a simple message as:

```
public void showMgs()
{
    System.out.println("Dreamtech Press");
 }
```

Now, convert the preceding method into a lambda expression by removing the public access specifier, return type declaration, such as void, and the name of method 'showMsg'. The lambda expression is shown as follows:

```
() -> {System.out.println("Dreamtech Press"); }
```

The basic structure of a lambda expression comprises:

- formal parameters separated by commas and enclosed in parentheses.
- arrow token, ->
- a single expression or a statement block

If a single expression is specified, the Java runtime environment automatically evaluates the expression and then returns its value. You can also use a return statement, but it should be enclosed in braces as it is not an expression.

You know that everything is treated as object in Java; therefore, objects are in large numbers therein. Performing operations on such large number of objects is very difficult, to handle which Java provides java.util package in which various objects are stored as a single object, known as collection object. You can perform operations on various objects by using the loops such as for-each loop. However, this loop will allow you to handle the object separately and not collectively. Therefore, you need to improve the functional aspect, which can be easily done by using lambda expressions as these are treated as objects by the JVM.

Method References

As discussed earlier, you can use lambda expressions for creating anonymous methods. Sometimes, a lambda expression is used only for calling an existing method by using its name but not for performing operations. Lambda expressions use method references that are compact and easy-to-read for calling existing methods by their names. There are mainly four different types of method references, as shown in Table 5.1:

Table 5.1: Different types of method references	
Types of References	**Example**
Referencing a static method	`ContainingClass::staticMethodName`
Referencing an instance method of a particular object	`containingObject::instanceMethodName`
Referencing an instance method of an arbitrary object of a particular type	`ContainingType::methodName`
Referencing to a constructor	`ClassName::new`

Now that you've the concepts behind inheritance, interfaces, inner classes, and lambda expressions, it's time to turn to the "Immediate Solutions" section.

Immediate Solutions

Creating a Subclass

"OK," the NP says, "I want to learn what inheritance is all about. Can you explain it in two words or less?" Counting on your fingers you say, "No way." Here's an example showing how to create a subclass by using inheritance.

Suppose you have a class `Airplane` that has one method `fly()`, which signifies that the object can fly and prints "Flying":

```
class Airplane { public void fly() { System.out.println("Flying");  }  }
```

Different kinds of airplanes fly in the sky, so if you want to specialize the `Airplane` class into a `Passengerplane` class, you can use inheritance. Java provides the `extends` keyword to use inheritance. Here's how you declare `Passengerplane` as a subclass of the `Airplane` class:

```
class Passengerplane extends Airplane {  . . . }
```

This code line signifies that the class `Passengerplane` is derived from the superclass `Airplane`, which means that the `Passengerplane` class will inherit the `fly()` method from the `Airplane` superclass. Subclasses can also contain your own methods and data members.

Here's an example in which we add a method named `land()` to the `Passengerplane` class that prints "Landing":

```
class Passengerplane extends Airplane {
   public void land() { System.out.println("Landing"); }
}
```

Now, we can access both the `fly()` method and the `land()` method in objects of the `Passengerplane` class, as shown in this example:

```
public class App {
         public static void main(String[] args) {
   Passengerplane p = new Passengerplane(); p.fly(); p.land(); }
}
```

Here's the output of the preceding code:

```
C:\>java App
Flying
Landing
```

This is a basic subclassing example. However, there is a lot more to the process. For example, what if you define a method in a subclass with the same name as a method in the superclass? How do you pass data to a superclass's constructor? All this is coming up in this chapter.

Access Specifiers and Inheritance

"Java's gone all wacky again," says the NP. "I want to use some methods from the superclass in a subclass, but Java says they don't even exist!" "What's the access specifier for the methods?" you ask. "Private," the NP says. "That's your problem," you say.

You use access specifiers with classes, data members, and methods to specify the visibility of those items in the rest of the program. Here's the general form of a class declaration and definition, showing how to use access specifiers:

```
access class classname [extends ...] [implements ...] {
   [access] [static] type instance_variable1;
   . . .
         [access] [static] type instance_variableN;
         [access] [static] returntype method1 (parameter_list) { . . . }
         . . .
         [access] [static] returntype methodN (parameter_list) { . . . }
}
```

You can find the details in Table 5.2. For more on access specifiers, please refer previous chapter.

Table 5.2: Scope by access specifier (x means "in scope")				
Location	**Private**	**No Modifier**	**Protected**	**Public**
Same class	x	x	x	x
Subclass in the same package		x	x	x
Non-subclass in the same package		x	x	x
Subclass in another package			x	x
Non-subclass in another package				x

For example, take a look at the NP's code, where the `fly()` method is not only declared `private` but also accessed in `main()`:

```
class Airplane {
   private void fly() { System.out.println("Flying"); }
}
class Passengerplane extends Airplane {
   public void land() { System.out.println("Landing"); }
}
public class App {
   public static void main(String[] args) {
   Passengerplane p = new Passengerplane();
   p.fly();  p.land();  }
}
```

Because declaring a member by using `private` restricts that member to its class, Java says it can't find the `fly` method as used in `main`:

```
App.java:10: cannot find symbol
     p.fly(); p.land();  }
      ^
  symbol :  method fly()
  location: variable p of type Passengerplane
1 error
```

On the other hand, declaring a member `protected` restricts its scope to code in the same package and subclasses of the class it's declared in. Therefore, the following code works:

```
class Airplane {
   protected void fly() { System.out.println("Flying"); } }

class  Passengerplane extends Airplane {
        public void land() { System.out.println("Landing"); }
}

public class App {
        public static void main(String[] args) {
   Passengerplane p = new Passengerplane (); p.fly(); p.land(); }
}
```

Here's the output of this application:

```
C:\>java App
Flying
Landing
```

Calling Superclass Constructors

"OK," says the NP, "I have come up with a problem. I know I can create a subclass from a superclass, but what if the superclass has a constructor? Does it ever get called?" "That," you say, "takes a little thought." "Hmm," says the NP, thinking.

Suppose you have a class named `Abc` that contains a constructor with no parameters:

```
class Abc {
   Abc() { System.out.println("In Abc\'s constructor"); }
}
```

Then you derive a subclass `Xyz` from `Abc`:

```
class Xyz extends Abc { }
```

Now, when you create an object of class `Xyz`, the constructor in class `Abc` is called automatically:

```
  public class App {
        public static void main(String[] args) { Xyz obj = new Xyz(); }
}
```

Here's the result of the preceding code:

```
C:\>java App
In Abc's constructor
```

Suppose you add a constructor to class `Xyz` that takes no parameters:

```
class Abc {
        Abc(){ System.out.println("In Abc\'s constructor"); }
   }
class Xyz extends Abc {
        Xyz() { System.out.println("In Xyz\'s constructor"); }
}
public class App {
        public static void main(String[] args) { Xyz obj = new Xyz(); }
}
```

In this case, when you instantiate class `Xyz`, the constructors from both classes `Abc` and `Xyz` are called:

```
C:\>java App
In Abc's constructor
In Xyz's constructor
```

Suppose you change `Xyz`'s constructor such that it takes only one parameter:

```
class Abc {
Abc() { System.out.println("In Abc\'s constructor"); }
}
class Xyz extends Abc {
        Xyz(String s) {
                System.out.println("In Xyz\'s String constructor");
  System.out.println(s); }
}
public class App {
        public static void main(String[] args) {
  Xyz obj = new Xyz("Hello Java!"); }
}
```

In this case, the constructors of both classes `Abc` and `Xyz` are called:

```
C:\>java App
In Abc's constructor
In Xyz's String constructor
Hello Java!
```

A constructor without any parameter is called a *default constructor* because Java calls it automatically when you create an object of any subclass of that class.

What does this mean? Suppose you add another constructor to `Abc` to handle strings as well (this is called *overloading*, and you'll see it later in this chapter):

```
class Abc {
        Abc() { System.out.println("In Abc\'s constructor"); }
        Abc(String s) {
  System.out.println("In Abc\'s String constructor");
  System.out.println(s); }
}
```

Suppose you don't want to call default constructor this time, instead you want the constructor with String parameter in `Abc` gets called.

```
class Abc {
  Abc() { System.out.println("In Abc\'s constructor"); }
        Abc(String s) { System.out.println("In Abc\'s String constructor");
System.out.println(s); }
}
class Xyz extends Abc {
```

```
            Xyz(String s) {
                    super(s);
      System.out.println("In Xyz\'s String constructor");
      System.out.println(s); }
    }
    public class App {
            public static void main(String[] args) {
      Xyz obj = new Xyz("Hello Java!"); }
    }
```

In this case, when you instantiate class `Xyz`, the constructor that takes a `String` parameter in class `Abc` is called instead of the default constructor:

```
C:\>java App
In Abc's String constructor
Hello Java!
In Xyz's String constructor
Hello Java!
```

Why is the constructor that takes a `String` parameter in class `Abc` called and not the default constructor in class `Abc`? The reason is simple—Java sees you are using the `super()` method to call the superclass's constructor:

```
class Xyz extends Abc {
            Xyz(String s) {
      super(s);
      System.out.println("In Xyz\'s String constructor");
      System.out.println(s); }
    }
```

If you use `super()` method to call a superclass's constructor, the line in which you do so must be the very first one in a constructor, which is where Java will look for it (if it's in any line but the first, Java will object).

Creating Multilevel Inheritance

"So," says the NP, "can I subclass subclasses?" "Yes," you say. "And can I subclass subclasses of subclasses?" "Yes," you say. The NP is prepared to go further and asks, "And can I…" You say, "How about some coffee?"

In the first topic of this chapter, we have created a class named `Airplane` and a subclass of `Airplane` named `Passengerplane`:

```
class Airplane {
        public void fly() { System.out.println("Flying"); }
   }
class Passengerplane  extends Airplane {
        public void land() { System.out.println("Landing"); }
}
```

That class hierarchy only included two levels—the superclass and the subclass—but things can go deeper. Say, for example, we have a new class `Fighterplane`, that has a subclasses `Airplane`, and adds a method named `fight`:

```
class Fighterplane extends Airplane {
        public void fight() { System.out.println("Fighting"); }
}
```

There are different kinds of aircraft, and we'll derive two further classes from `Fighterplane`—`Hawk`, which defines a new method named `search`, and `Jet`, which defines a method named `target`:

```
class Hawk extends Fighterplane {
        public void search() { System.out.println("Searching"); }
}
class Jet extends Fighterplane {
        public void target() { System.out.println("Targeting"); }
}
```

This is what the class hierarchy now looks like:

```
Airplane
   |_____ Passengerplane
          |_____ Fighterplane
                 |_______ Hawk
                 |_______Jet
```

Now, we can instantiate objects of these classes, as in this example, we are creating objects of the classes `Passengerplane` and `Jet`:

```
public class App {
        public static void main(String[] args) {
  System.out.println("Creating a passenger plane...");
  Passengerplane p = new Passengerplane(); p.fly(); p.land();
  System.out.println("Creating a jet...");
  Jet j = new Jet();
  j.fly(); j.fight(); j.target();
}
}
```

Here's what the output of this code looks like:

```
C:\>java App
Creating a passenger plane...
Flying
Landing
Creating a jet...
Flying
Fighting
Targeting
```

Handling Multilevel Constructors

"Hey!" the NP says. "Has Java gone all wacky again?" you ask. "Yes," the NP says, "I have created four levels of subclasses, classes `A` through `D`, where `A` is the superclass everything is derived from. It makes sense that `D`'s constructor should be called first, then `C`'s, then `B`'s, and then `A`'s, right?" "No," you say, "which is why Java calls them in exactly the opposite order." "I knew you'd be on my side," the NP says.

Let's look at an example of multilevel constructor use. Here, we'll implement the NP's program with four levels of subclassing, starting with class `A`:

```
class A {
A() { System.out.println("Constructing A..."); }
}
class B extends A {
B() { System.out.println("Constructing B..."); }
}
class C extends B {
C() { System.out.println("Constructing C..."); }
}
class D extends C {
D() { System.out.println("Constructing D..."); }
}
```

Next, we'll create an object of class `D`, the last class in the subclassing chain:

```
public class App {
public static void main(String[] args) { D obj = new D(); }
}
```

Here's what you see when you run this code:

```
C:\>java App
Constructing A...
Constructing B...
Constructing C...
Constructing D...
```

In other words, Java called `A`'s constructor first, then `B`'s, then `C`'s, then `D`'s, not in the reverse order as you might expect. Why does Java do it this way? Because when you create subclasses, you proceed from the general to the specific, which means that class `A` knows nothing about class `B`, class `B` knows nothing about class `C`, and so on. For that reason, Java calls the original subclass's constructor first, then the next, and so on. Because class `B` knows about class `A`, it might rely on certain parts of `A` being initialized before completing its own initialization, and the same for class `C` with respect to class `B`, and so on.

It's also worth noting that you can pass parameters back multiple levels in the ways we outlined in the topic "Calling Superclass Constructors," earlier in this chapter. However, all constructors in the subclassing chain must still be called in an ascending order.

Overriding Methods

"Oh well," the NP says, "I thought I could use the Java `Button` class in my new program, but I wanted to create a method named `getLabel()`, and the `Button` class already has a method named that." "That's no problem," you say, "you can just override the `Button` class's `getLabel()` method with a new implementation of that method." "I can do that?" the NP asks.

In the last chapter, you saw that you can overload methods with different implementations that have different parameter lists. You can also *override* methods that you inherit from a superclass, which means that you replace them with a new version.

Here's an example. In this case, we'll start with a general base class named `Animal` that has one method `breathe()`. When `breathe()` is called, it prints out "Breathing." Here's the code:

```
class Animal {
        public void breathe() { System.out.println("Breathing"); }
}
```

Now, suppose you want to derive a new class from `Animal` named `Fish`. When you test the `breathe()` method in the `Fish` class, you see that it prints out "Breathing." You decide it would be better if it prints out "Bubbling" instead. To do this, you can override the `breathe()` method in the `Fish` class simply by defining a new version with the same parameter list:

```
class Animal {
        public void breathe() { System.out.println("Breathing"); }
}
class Fish extends Animal {
        public void breathe() { System.out.println("Bubbling"); }
}
```

Now, you can instantiate new objects of the `Animal` and `Fish` classes and call their `breathe` methods as follows:

```
public class App {
        public static void main(String[] args) {
  System.out.println("Creating an animal...");
  Animal a = new Animal(); a.breathe();
  System.out.println("Creating a lungfish...");
  Fish f = new Fish(); f.breathe(); }
}
```

Here's the output of this code, showing that the `breathe()` method is indeed overloaded:

```
C:\>java App
Creating an animal...
Breathing
Creating a lungfish...
Bubbling
```

You can find the process of creating methods in detail in Chapter 4.

Accessing Overridden Members

"Well," the NP says, "I think I have run into a problem that no one else has ever encountered in Java." "Oh yes?" you ask. "Yes," the NP says, "I have overridden a superclass's method and that's fine most of the time, but sometimes, I need access to the original overridden method." "That's a common problem," you say, "and you can solve it with the `super` keyword."

You can use `super` much like you use the `this` keyword, except that `super` doesn't refer to the current object but rather to its superclass. For example, take a look at this code from the previous topic in which the `Fish` class subclasses the `Animal` class and overrides the `breathe()` method so that it prints out "Bubbling" instead of "Breathing":

```
class Animal {
```

```
        public void breathe() { System.out.println("Breathing"); }
}
class Fish extends Animal {
        public void breathe() { System.out.println("Bubbling"); }
}
```

Now, however, suppose you realize that a certain type of fish—a lungfish, for example—can indeed breathe as land animals do. So, you add a new method to the `Fish` class, `newbreathe()`. In this method, you'd like to reach the superclass's `breathe()` method, and you can do that with the `super` keyword like this:

```
class Animal {
        public void breathe() { System.out.println("Breathing"); }
}
class Fish extends Animal {
        public void breathe() { System.out.println("Bubbling"); }
                public void newbreathe() { super.breathe(); }
}
```

Now, you can instantiate objects of the `Animal` and `Fish` classes and use the `newbreathe()` method like this:

```
public class App {
  public static void main(String[] args) {
  System.out.println("Creating an animal...");
  Animal a = new Animal(); a.breathe();

  System.out.println("Creating a lungfish...");
  Fish lf = new Fish(); lf.newbreathe(); }
}
```

Here's the result of this code:

```
C:\>java App
Creating an animal...
Breathing
Creating a lungfish...
Breathing
```

Using Superclass Variables with Subclassed Objects

The Programming Correctness Czar (PCC) appears and says, "In C++, you can assign a subclass object reference to a variable of a superclass type. Can you do that in Java?" The says, "Run that by me again?" "Yes," you say in response to the PCC.

One interesting aspect of OOP in Java—which we'll put to work in the next topic—is that a subclass object's reference can be assigned to a reference variable of superclass. For example, class `Abc` is a superclass of `Xyz` and you have a variable of `Abc`, then the reference of `an` object of class `Xyz` can be stored in a variable of type `Abc`.

Let's take a look at an example. Here, we'll use the multilevel class hierarchy you saw earlier in this chapter:

```
class Airplane {
        public void fly() { System.out.println("Flying"); }
}
class Passengerplane extends Airplane {
        public void land() { System.out.println("Landing"); }
}
class Fighterplane extends Airplane {
        public void fight() { System.out.println("Fighting"); }
}
class Hawk extends Fighterplane {
        public void search() { System.out.println("Searching"); }
}
class Jet extends Fighterplane {
        public void target() { System.out.println("Targeting"); }
}
```

For example, to create new objects of the `Passengerplane` and `Jet` classes, we can use this code:

```
public class App {
        public static void main(String[] args) {
  System.out.println("Creating a fighterplane...");
```

```
    Fighterplane fp = new Fighterplane();
    fp.fly(); fp.fight();
  System.out.println("Creating a jet...");
    Jet j = new Jet(); j.fly(); }
}
```

However, we can also assign the new `Jet` object to a variable of class `Airplane` as follows:

```
public class App {
        public static void main(String[] args) {
                System.out.println("Creating a fighterplane...");
                Fighterplane fp = new Fighterplane();
                fp.fly(); fp.fight();
                System.out.println("Creating a jet...");
                Airplane a = new Jet(); a.fly();
                //j.fight();
                //j.target();
        }
}
```

Here's the output of this code:

```
C:\>java App
Creating a fighterplane...
Flying
Fighting
Creating a jet...
Flying
```

Note that we commented out the lines `j.fight()` and `j.target()` here because those methods are defined in the `Fighterplane` and `Jet` classes, which are subclasses of `Airplane`, which means that those methods can't be used with a variable of class `Airplane`.

Generally, the object variable `a` will only allow access to items present in its own class. It'll not allow access to any member that doesn't belong to the class `Airplane`.

There's a use for this seemingly arcane piece of information—see the next topic for the details.

Dynamic Method Dispatch (Runtime Polymorphism)

"Hmm," says the NP, "I have another problem. My drawing program can create objects of the classes' `Triangle`, `Square`, and `Circle`, each of which has a `draw()` method, but I'm not sure what type of object the user will want to create until the program runs. Will I have to write the program's main code three times, one for each type of object?" "Not at all," you say. "You can use runtime polymorphism." "Huh?" the NP replies.

Runtime polymorphism or *dynamic method dispatch* lets you wait until the program gets executed before deciding the object whose reference is to be stored.

This means, in the NP's case, that he can write his code calling the `draw()` method on various variables and decide what type of object—`Triangle`, `Square`, or `Circle`—is stored in those variables at runtime.

As discussed in the previous topic, you can assign a subclass object reference to a superclass type variable. You may think that Java is a strictly typed language, then why does it allow this? The answer is to support runtime polymorphism.

Here's an example to make this clearer. In this case, we'll create a superclass named `W`, a subclass of `W` named `X`, a subclass of `X` named `Y`, and a subclass of `Y` named `Z`, each of which has a `print()` method:

```
class W {
        public void print() { System.out.println("Here's W"); }
}
class X extends W {
        public void print() { System.out.println("Here's X"); }
}
class Y extends X {
        public void print() { System.out.println("Here's Y"); }
}
class Z extends Y {
```

```
        public void print() { System.out.println("Here's Z"); }
}
```

Now, we can create an object reference of each class type:

```
public class App {
        public static void main(String[] args) {
  W w1 = new W(); X x1 = new X();
  Y y1 = new Y(); Z z1 = new Z();
  . . .
```

To show how runtime polymorphism works, we'll also create a variable named `ref` that holds an object reference to an object of class `W`:

```
public class App {
        public static void main(String[] args) {
  W w1 = new W(); X x1 = new X();
  Y y1  = new Y(); Z z1 = new Z(); W ref;
  . . .
```

We can store references of objects of different classes in `ref`, and upon calling, the `print()` method of the corresponding class is called:

```
public class App {
        public static void main(String[] args) {
  W w1 = new W(); X x1 = new X(); Y y1 = new Y();
  Z z1 = new Z();
  W ref; ref = w1; ref.print(); ref = x1; ref.print();
  ref = y1; ref.print(); ref = z1; ref.print(); }
}
```

Here's the result of this code:

```
C:\>java App
Here's W
Here's X
Here's Y
Here's Z
```

Using runtime polymorphism, you can write a code that will work with many different types of objects and decide on the actual object type at runtime. Note that the restrictions mentioned in the previous topic still apply. The object variable `a` will only allow access to items present in its own class. It'll not allow access to any member that doesn't belong to class `A`.

Creating Abstract Classes

The NP stomps in. "That darn Johnson!" the NP says. "What's wrong?" you ask. "That darn Johnson was using one of my classes that needed to be customized before you can use it but that darn Johnson didn't. So, it didn't work—right there in front of the Big Boss!" "Well," you say, "next time, make your class abstract and that darn Johnson will *have* to customize it."

While writing a class, you may find cases where you can only provide general code, and it's up to the developer who subclasses your class to customize it. To make sure the developer customize your code, you can make the method *abstract*, which means the developer will have to override your method; otherwise, Java will complain. To make a method abstract, you use the `abstract` keyword. If you make any methods in a class abstract, you also have to make the class itself abstract as well.

Here's an example. In this case, we'll create a class named `A` that has a method named `print()`, which prints out a string, and it gets the string to print by calling a method named `getData()`:

```
class A {
        String getData(); public void print() { System.out.println(getData()); }
}
```

Note that there's no implementation of the `getData()` method because we want developers to specify what data they want to print out. To make sure they know that they must provide an implementation of the `getData()` method, we can make the method abstract, which means we must make the class itself abstract as well:

```
abstract class A {
        abstract String getData();
```

```
        public void print() { System.out.println(getData()); }
}
```

Now, when we subclass `A`, we have to provide an implementation of `getData()` like this:

```
class B extends A {
        String getData() { return "Hello from Java!"; }
}
```

Here's how we can put the subclass to work (note that an abstract class cannot be instantiated directly):

```
public class App {
        public static void main(String[] args) { B b1 = new B(); b1.print(); }
}
```

Here's the result of this code:

```
C:\>java App
Hello from Java!
```

TIP

This is an important technique to understand because many methods in the Java packages themselves are abstract and therefore must be overridden.

Stopping Overriding with **final**

The NP says, "That darn Johnson!" "What's wrong?" you ask. "That darn Johnson overrode the `draw()` method in my `painter` class and messed it all up," the NP complains. "Don't worry, NP," you say, "you can mark that method as *final*, and no one can override it then." "Swell!" says the NP.

Earlier in this chapter, you saw an example in which the class `Fish` overrode the method called `breathe()` in the `Animal` class:

```
class Animal {
        void breathe() { System.out.println("Breathing"); }
}
class Fish extends Animal {
        public void breathe() { System.out.println("Bubbling"); }
}
```

If for some reason you don't want to let anyone override the `breathe()` method, you can declare it `final` as follows:

```
class Animal {
        final void breathe() { System.out.println("Breathing"); }
}
class Fish extends Animal {
        public void breathe() { System.out.println("Bubbling"); }
}
```

Now, let's say you try to use these classes in some code:

```
public class App {
        public static void main(String[] args) {
   System.out.println("Creating an animal...");
   Animal a = new Animal(); a.breathe();
   System.out.println("Creating a lungfish...");
   Fish f = new Fish(); f.breathe(); }
}
```

Java will object that you can't override `breathe()` like this:

```
C:\>javac App.java
App.java:5: error: breathe() in Fish cannot override breathe() in Animal
            public void breathe() { System.out.println("Bubbling"); }
                        ^
  overridden method is final
1 error
```

Stopping Inheritance with **final**

You can prevent a class from being subclassed by declaring the entire class `final`, as you see here:

```
final class Animal {
```

```
        public void breathe() { System.out.println("Breathing"); }
}
class Fish extends Animal {
        public void breathe() { System.out.println("Bubbling"); }
}
public class App {
        public static void main(String[] args) {
   System.out.println("Creating an animal...");
   Animal a = new Animal(); a.breathe();
   System.out.println();
   System.out.println("Creating a lungfish...");
   Fish f = new Fish(); f.breathe(); }
}
```

Here's what happens when you try to execute this code:

```
C:\>javac App.java
App.java:4: error: cannot inherit from final Animal
class Fish extends Animal {
                   ^
1 error
```

Creating Constants with **final**

In the previous two topics, we showed two uses for `final`: to prevent method overriding and subclassing. There's another use for `final` in Java—you can use it to declare constants.

Here's an example in which we make a variable into a constant with `final` and then try to assign a value to it:

```
public class App {
        public static void main(String[] args) { final int a = 5; a = 6; }
}
```

Here's how the Java compiler objects to this code:

```
C:\> javac App.java
App.java:2: error: cannot assign a value to a final variable a
             public static void main(String[] args) { final int a = 5; a = 6; }
                                                                          ^
1 error
```

Is-a vs. Has-a Relationships

The PCC arrives and says, "In C++, you can have is-a relationship and has-a relationship in inheritance." "Same in Java," you say.

You may come across the terms *is-a* and *has-a* when working with inheritance because they specify two of the ways the classes can relate to each other. Standard inheritance is what you usually think of in terms of an *is-a* relationship, as in the following example. In this case, class `A` extends class `B`, so you can say class `a` *is-a* `b`:

```
class A extends B {
        A() { print(); }
}
class B {
        void print() { System.out.println("This comes from class B..."); }
}
public class App {
        public static void main(String[] args) { A obj = new A(); }
}
```

When you call `A`'s `print()` method, you are actually calling the print() method `A` inherited from `B`, and it's this method that does the printing:

```
C:\>java App
This comes from class B...
```

In a *has-a* relationship, on the other hand, one object includes an object reference to another, as in this case, where objects of class `A` will include an internal object of class `B`:

```
class A {
        B b1;
        A() { b1 = new B(); b1.print(); }
```

```
}
class B {
        void print() { System.out.println("This comes from class B..."); }
}
public class App {
        public static void main(String[] args) { A obj = new A(); }
}
```

Now, class `B`'s print() method is accessible from the object named `b1` in the object of class `A`:

```
C:\>java App
This comes from class B...
```

The Java **Object** Class

"All I want," the NP says, "is to write a simple class, no inheritance whatsoever and. . ." "Too late," you say. "All classes in Java are subclasses of one master class: `java.lang.Object`, so anything you do already involves inheritance."

Every class in Java is derived automatically from the `java.lang.Object` class, and there are certain advantages to know this, including knowing that all objects have already inherited quite a few methods, ready for you to use. The methods of class `Object` appear in Table 5.3:

Table 5.3: The Java object class methods

Method	Does this
`protected Object clone()`	It yields a copy of this object
`boolean equals(Object obj)`	It indicates whether another object is equal to this one
`protected void finalize()`	It is called by the garbage collector on an object when garbage collection is about to dispose of the object
`Class<?> getClass()`	It yields the runtime class of an object
`int hashCode()`	It yields a hash code value for the object
`void notify()`	It wakes up a single thread that's waiting on this object's monitor
`void notifyAll()`	It wakes up all threads that are waiting on this object's monitor
`String toString()`	It yields a string representation of the object
`void wait()`	It makes the current thread wait until another thread invokes the `notify` method or the `notifyAll` method
`void wait(long timeout)`	It makes the current thread wait until either another thread invokes the `notify` method or the `notifyAll` method or a specified amount of time has passed
`void wait(long timeout, int nanos)`	It causes the current thread to wait until another thread invokes the `notify` method or the `notifyAll` method for this object, some other thread interrupts this thread, or a certain amount of real time has passed

Here's an example in which we use the `getClass()` method to determine the class of an object reference in a superclass variable. This is useful because a superclass variable can hold references to objects of any of its subclasses. We start with a superclass named `A` and three subclasses `B`, `C`, and `D`. The `print()` method in each of these classes prints out the name of the class. Here's the code:

```
class A {
        public void print() { System.out.println("Here's A..."); }
}
class B extends A {
        public void print() { System.out.println("Here's B..."); }
}
class C extends A {
        public void print() { System.out.println("Here's C..."); }
}
class D extends A {
```

```
        public void print() { System.out.println("Here's D..."); }
}
```

Next, we create an instance of each class and a variable of class `A` named `aref`:

```
public class App {
        public static void main(String[] args) {
  A a1 = new A(); B b1 = new B(); C c1 = new C();
  D d1 = new D(); A aref;
  . . .
```

Now we can determine the class of the object in `aref`, no matter which of the subclasses it is:

```
public class App {
        public static void main(String[] args) {
  A a1 = new A(); B b1 = new B(); C c1 = new C();
  D d1 = new D(); A aref;
  aref = a1;
  System.out.println("aref\'s class is now " + aref.getClass());
  aref.print(); aref = b1;
  System.out.println("aref\'s class is now " + aref.getClass());
  aref.print(); aref = c1;
  System.out.println("aref\'s class is now " + aref.getClass());
  aref.print(); aref = d1;
  System.out.println("aref\'s class is now " + aref.getClass());
  aref.print(); }
}
```

Here's the result:

```
C:\>java App
aref's class is now class A
Here's A...
aref's class is now class B
Here's B...
aref's class is now class C
Here's C...
aref's class is now class D
Here's D...
```

As you can see, each object's built-in methods—such as `getClass()`—can be very useful.

Using Interfaces for Multiple Inheritance

"Hmm," says the NP, "I have a class named `Animal` and an entirely separate class named `Mineral`, and I want to inherit from both of those classes at the same time to create something entirely new." "Sorry," you say, "Java doesn't support multiple inheritance directly—you'll have to make one of those classes an *interface*."

In other languages, such as C++, one class can inherit from multiple classes at once, but this technique doesn't work directly in Java—that is, you can only use the `extends` keyword with one class at a time. Therefore, you can't do this:

```
class A extends B, C //Won't work!
{ . . . }
```

However, there are two ways to implement what amounts to multiple inheritance in Java. The first is to use single inheritance in stages (if that will work for the classes you want to inherit from) as follows:

```
class C { . . . }
class B extends C { . . . }
class A extends B { . . . }
```

The other way is to use interfaces. Interfaces will start becoming important in the next chapter, so we'll introduce them now.

An *interface* specifies the form of its methods but does not give any implementation details; therefore, you can think of it much like the declaration of a class. As you'll see later in the book, you can create interfaces with the `interface` keyword:

```
interface C { . . . }
interface B { . . . }
```

Now, you can use these two interfaces with the `implements` keyword:

```
class A implements B, C { . . . }
```

As the interfaces declare and not define methods, it's up to you to implement the interfaces' methods. For example, in the applet shown at the beginning of this chapter, we implemented the Java `ActionListener` interface to handle button clicks (you'll see all the details of applets like this in the next chapter). That interface declares one method `actionPerformed()`, which we defined as follows:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Clicker extends Applet implements ActionListener {
         TextField text1;
         Button button1;
         public void init() {
   text1 = new TextField(20); add(text1);
   button1 = new Button("Click Here!"); add(button1);
   button1.addActionListener(this); }
         public void actionPerformed(ActionEvent event) {
   String msg = new String ("Welcome to Java");
   if(event.getSource() == button1) { text1.setText(msg); } }

}
```

If you don't define the `actionPerformed()` method, the Java compiler will give you a message like this one:

```
C:\>javac Clicker.java
Clicker.java:4: error: Clicker is not abstract and does not override abstract method
  actionPerformed(ActionEvent) in ActionListener
public class Clicker extends Applet implements ActionListener {
       ^
1 error
```

You can implement as many interfaces as you want, as in this example, which implements the `ActionListener` interface for button clicks and the `MouseListener` and `MouseMotionListener` interfaces to work with the mouse:

```
import java.awt.Graphics;
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
public class Dauber extends Applet implements ActionListener, MouseListener,
  MouseMotionListener  {
         Button buttonDraw, buttonLine, buttonOval, buttonRect,
button3DRect; Button buttonRounded; Point pts[] = new Point[1000];
         Point ptAnchor, ptDrawTo, ptOldAnchor, ptOldDrawTo; int ptindex = 0;
         . . .
```

The *Readable* Interface

One of the most dominant concepts that have been implemented by Java is interfaces. A popular misconception is that interfaces are Java's substitute for multiple inheritances. It can only have abstract methods and do not have a method body. It is like a class that can inherit from another interface but not from a class. Interfaces can only include final static public variables as all members of an interface are public. However, it need not be specified that they are abstract. Table 5.4 shows the single method contained in the `Readable` interface:

Table 5.4: Methods of Readable Interface

Method	Does this
int read(CharBuffer cb)	It attempts to read characters into the specified character buffer

The `Readable` interface is a source of characters and the `CharBuffer` class ensures the availability of characters to the callers from the `Readable` interface. So, the class `CharBuffer` should be inherited by the `Readable` interface as follows:

```
public abstract class CharBuffer extends Buffer
implements Comparable<CharBuffer>, Appendable, CharSequence, Readable
```

Considering another example, `InputStreamReader` is extended by the `Reader` class, as follows:

```
public class InputStreamReader extends Reader
```

The `InputStreamReader` class acts as a bridge from byte streams to character streams. This class reads bytes and then decodes them into characters with the help of a mentioned charset.

Now, going to the details of the `Reader` class. This class is inherited by the `Readable` interface, shown as follows:

```
public abstract class Reader extends Object implements Readable, Closeable
```

As the `InputStreamReader` class needs to use the `read()` method, it needs to be inherited by the `Readable` class. The `Readable` interface is implemented by a number of reader classes such as `BufferReader`, `CharBuffer`, `FileReader`, `InputStreamReader`, and `Scanner`.

The *Appendable* Interface

The interface to which `char` sequences and values can be tagged on is an `Appendable` interface. All classes intended to receive formatted output from a formatter must implement the `Appendable` interface. Only valid Unicode characters as described in Unicode Character Representation can be appended. You must take care to ensure that supplementary characters can contain multiple 16-bit `char` values. The `Appendable` interface and its `append()` methods are implemented in classes that support adding characters to the end of an object, typically a stream. Indirectly used through the `Formatter` class, you can attach a character, `CharSequence`, or `CharSequence` subset.

It is not necessary that appendables are always safe for multithreaded access. Basically, classes that extend and implement this interface are responsible for thread safety. However, there is no assurance that no errors will propagate to the invoker as this interface is implemented by existing classes with different styles of error handling. Table 5.5 shows the methods of the Appendable interface:

Table 5.5: Methods of Appendable interface

Method	Does this
`Appendable append(char c)`	It appends the specified character to this `Appendable`
`Appendable append(CharSequence csq)`	It appends the specified character sequence to this `Appendable`
`Appendable append(CharSequence csq, int start,int end)`	It appends a subsequence of the specified character sequence to this `Appendable`

Besides `PrintStream`, the interface is also implemented by `CharBuffer` and all `Writer` classes, i.e., among other things by `BufferedWriter`, `CharArrayWriter`, `FileWriter`, `FilterWriter`, `OutputStreamWriter`, and `StringWriter`.

From `PrintStream` and `Appendable`, the covariant return can be read-off. Before Java 5, the subclasses and/or the implementing classes had to adhere accurate with the type of return to the upper class or interface. That makes the `PrintStream` in such a way that it will not append in the class, the return `Appendable`, but with the return `PrintStream`, it will be `Appendable`.

```
public PrintStream append( char C ){ print(C); return this; }
```

The *Iterable* Interface

The `Iterable<>` interface is a standard interface type that declares a single method `Iterator()`, which in turn returns a reference of type `Iterator<>` and is another generic interface type. A `for-each` loop can be used to iterate through an object that implements this interface. Making a class iterable tells clients that they can iterate through its contents by using a `for-each` loop. Oracle has added a new interface to Java that allows you to mark your classes as iterable, this is the `java.lang.Iterable` interface. Table 5.6 shows the methods of the Iterable interface:

NOTE

Special care must be taken to note that Iterable is in java.lang, not in java.util. No explicit documentation about this is available, probably to avoid importing the interface (java.lang is in the set of namespaces automatically imported for all Java code).

interface java.long. Iterable<T>

Table 5.6: Methods of Iterable interface	
Method	**Does this**
`Iterator<T> iterator()`	It supplies an iterator, which iterates `T` over all elements of the type

Many classes have already implemented this interface so that the result quantities can be iterated with this extended interface. Primarily, it concerns with the data structures. In addition, it comes with the field, which is not directly visible as class, but can be `Iterable` suitably and be implemented.

Here's an example to implement a practical `Iterable` in order to go over characters in forward and backward directions. This example uses an `ArrayList` object.

```
import java.util.*;
class Iter {
        public static void main(String args[]) {
        ArrayList<String> al = new ArrayList<String>();
        al.add("A");
        al.add("B");
        al.add("C");
        al.add("D");
        al.add("E");
        al.add("F");
        System.out.print("Contents of it are: ");
        Iterator<String> Str = al.iterator();
        while(Str.hasNext()) {
  String element = Str.next();
  System.out.print(element + " "); }
        System.out.println();
        ListIterator<String> list_itr = al.listIterator();
        while(list_itr.hasNext()) {
  String element = list_itr.next();
  list_itr.set(element + "+"); }
        System.out.print("Contents of it after modification: ");
        Str = al.iterator();
        while(Str.hasNext()) {
        String element = Str.next();
        System.out.print(element + " "); }
        System.out.println();
        System.out.print("Contents of it in backward direction: ");
        while(list_itr.hasPrevious()) {
        String element = list_itr.previous();
        System.out.print(element + " "); }
        System.out.println(); }
}
```

Here's the result of the code:

```
C:\>java Iter
Contents of it are: A B C D E F
Contents of it after modification: A+ B+ C+ D+ E+ F+
Contents of it in backward direction: F+ E+ D+ C+ B+ A+
```

Here's another example to implement a practical `Iterable` in order to go over words of a sentence. Fundamental implementation of the `Iterable` interface on array of characters serves the purpose of the `for-each` loop and decides about whether characters of string can be selected or not.

We begin with the first part to be create the class `App`, which must implement `Iterable`, as shown in the code below:

```
import java.util.*;
import java.util.Iterator;
public class App implements Iterable<String> {
static ArrayList<String> astr = new ArrayList<String>();
public static void main( String args[] ) {
astr.add("At the beginning was the word - At the end the cliche");
Iterator<String> it = astr.iterator();
while(it.hasNext()) {
String element = it.next();
```

```
    System.out.println(element); }
    }
    public Iterator<String> iterator() { return astr.iterator(); }
    }
```

Here's the result of the code:

```
    C:\> Java App
    At the beginning was the word - At the end the cliche
```

Let's see one more example related to the `Iterable` interface:

```
    package java.lang;
    import java.util.Iterator;
    public interface Iterable<T> { Iterator<T> iterator(); }
```

To implement this interface in the `Catalog` class, you must write an `Iterator` method. This method must return an `Iterator` object that is bound to the product type.

The `Catalog` class stores all of its `Product` objects in an `ArrayList`. An `Iterator` object is provided to a client for returning the `Iterator` object from the products `ArrayList` itself. Here is the modified `Catalog` class:

```
    class Catalog implements Iterable<Product> {
            private List<Product> products = new ArrayList<Product>();
            void add(Product product) { products.add(product); }
    public Iterator<Product> iterator() { return products.iterator(); }
    }
```

You can see that the `Iterator`, the `List` reference, and the `ArrayList` are all bound to the `Product` type. After the implementation of the `java.lang.Iterable` interface, the `for-each` loop can be used by the client code.

Here is an example:

```
    Catalog cat = new Catalog();
      cat.add(new Product("1", "pinto", new BigDecimal("7.99")));
      cat.add(new Product("2", "flounder", new BigDecimal("34.88")));
      cat.add(new Product("2", "cucumber", new BigDecimal("9.01")));
      for (Product product: cat)
            product.discount(new BigDecimal("0.1"));
            for (Product product: cat)
            System.out.println(product);
```

Creating **Iterable** Objects

An object of a container class type is usable with the collection-based `for` loop only if it implements the generic `Iterable<>` interface defined in the `java.lang` package as follows:

```
    public interface Iterable<T>
```

The `Iterable<>` interface comprises a single method called `Iterator()`, which returns a reference of type `Iterator<>`. This method is a generic interface type. The implementation of this interface enables an object to be used by the "`for-each`" statement.

- **Iterator<T> iterator()** — Returns an Iterator over a group of elements of type T.
- **public interface Iterator<E>** — Refers to an Iterator over a collection that replaces Enumeration in the Java collections framework.

Iterators differ from enumerations in the following two respects:

- Iterators let the caller remove elements from the collection lying under, at the time of iteration with properly defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework. Consider the following code snippet that uses the `Iterator` interface:

```
    Iterator iter = ...;
    while (iter.hasNext()) { Object element = iter.next(); } // process element
```

Anything (including your own classes) that implements the `Iterable` interface can be used in a `for-each` loop. The `for-each` takes anything that has an `iterator()` method and iterates through the available elements. But, how do you find out that an `iterator()` method is present? The answer is that the new `Iterable` interface indicates the presence of the `iterator()` method. Let's have a look at the following example:

```
import java.util.ArrayList;
import java.util.Iterator;
public class App {
  public static void main(String[] args) {
        double[] arr = {1.5, 4.2, 4.7, 9.4, 4.8, 7.6};
        for(double d: arr) { System.out.println(d); }
        System.out.println("---------------------");
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(17); list.add(10); list.add(-61);
        for(Integer number: list) { System.out.println(number); }
  System.out.println("---------------------");
  for(int item: list) { System.out.println(item); }
        System.out.println("---------------------");
        MyIterable iter = new MyIterable();
        for(double d: iter) { System.out.println(d); }
        }
}
class MyIterable implements Iterable<Double>,
Iterator<Double> {
        static final double CLOSE_TO_ZERO = 0.0001;
        double value; double factor = -0.5;
        public MyIterable() { value = 1.0; }
        public Iterator<Double> iterator() { return this; }
        public boolean hasNext() {
        return !(value > -CLOSE_TO_ZERO && value
        < CLOSE_TO_ZERO); }
        public Double next() {
        double res = value; value = value*factor; return res; }
        public void remove() {
        throw new UnsupportedOperationException("There is
        nothing to remove"); }
}
```

Here's the result of this code:

```
C:\>java App
1.5
4.2
4.7
9.4
4.8
7.6
---------------------
17
10
-61
---------------------
17
10
-61
---------------------
1.0
-0.5
0.25
-0.125
0.0625
-0.03125
0.015625
-0.0078125
0.00390625
```

```
-0.001953125
9.765625E-4
-4.8828125E-4
2.44140625E-4
-1.220703125E-4
```

Creating Inner Classes

"OK," says the NP, "I'm an expert now. Is there anything about classes that I don't know?" You smile and say, "Plenty. For example, what do you know about inner classes?" "What classes?" the NP asks.

Beginning in Java 1.1, you could nest class definitions inside each other. Nested classes can be static or nonstatic; however, static classes cannot refer to the members of its enclosing class directly, they must instantiate and use an object instead, so they are not often used. *Inner classes*, on the other hand, are nonstatic nested classes, and they are quite popular for reasons having to do with event handling, which we'll get into in the next chapter.

Here's an example of an inner class; in this case, class `B` is defined inside class `A`, and we instantiate an object of class `B` in class `A`'s constructor:

```
class A {
        B obj;
        A() {
                obj = new B();
                obj.print(); }
  class B {
        public void print() { System.out.println("Inside B..."); }
        }
}
public class App {
public static void main(String[] args) { A obj = new A(); }
}
```

When this code runs, it instantiates an object of class `A`, which instantiates an internal object of class `B` and calls that object's `print()` method. Here's the result:

```
C:\>java App
Inside B...
```

Besides inner classes, as demonstrated in this example, you can also have *anonymous* (unnamed) inner classes—see the next topic for the details.

Creating Anonymous Inner Classes

One type of shorthand that's handy for working with event handling is to use *anonymous inner classes*. An anonymous inner class is one that doesn't have any name, and you create it by using this syntax:

```
new SuperType(constructor parameters) { } //methods and data
```

In the preceding syntax, `SuperType` is the name of a class or interface you are subclassing (you must specify a superclass type when creating anonymous inner classes), and you can define the anonymous inner class's methods and data in a code block.

Let's look at an example. In this case, we'll create an anonymous inner class inside a new class—class `A`. This anonymous inner class will subclass another class, class `B`, and defines a method named `print()`, which we call immediately, like this:

```
class A {
        A() {
  (new B() {public void print() {
  System.out.println("Hello from Java!"); }}).print();
  }
}
class B{ }
```

All that's left is to create an object of class `A` and to call that class's constructor, which we do like this:

```
public class App {
        public static void main(String[] args) { A obj = new A(); }
}
```

Here's the result of this code:

```
C:\>java App
Hello from Java!
```

Using Lambda Expression

After writing a lambda expression, you can call and execute it like a method. You can call a lambda expression by creating a functional interface, which is an interface comprising only one abstract method. However, it may have multiple static or default methods. The advantage of using functional interface is that it can be used for assigning a lambda expression or as a method reference. Consider the following example in which a lambda expression is created and then it is assigned to an object of a functional interface:

```
class LambdaExpression
{
   //functional interface
   interface FuncInterface
   {
      void showMsg();
   }
    public static void main(String args[])
   {
            FuncInterface Fi = () -> { System.out.println("Hello Dreamtech Press"); };

      Fi.showMsg();

   }
}
```

Here is the result of the preceding code:

```
C:\>java LambdaExpression
Hello Dreamtech Press
```

Default Methods

A method that is declared by using the default keyword is known as default method. It is accessible by all implementation classes of that interface. These classes can override the implementation of the default method and provide a different implementation. The concept of default method is new, and it is an important feature included in Java SE 8.0. The default method also allows adding new functionality to interfaces present in libraries and makes sure that new code written for these interfaces should be compatible with their older versions. Consider the following example in which a default method is created with the name div and is called using the object of the interface:

```
//Functional interface with a default and an abstract methods
interface FuncInterface
{
    //abstract method
    void mul(int num1, int num2);
    //default method
    default void div(int num1, int num2)
    {
        int division=num1/num2;
  System.out.println("Result of Division: "+division);
    }
}
class ImplClass implements FuncInterface
{
    public void mul(int num1, int num2)
   {
  int sum=num1+num2;
  System.out.println("Result of Addition: "+sum);
   }
}
class LambdaExpression1
{
   public static void main(String args[])
```

```
      {
       FuncInterface Fi = new ImplClass();
       Fi.mul(2,3);
       Fi.div(15,3);
    }
    }
```

Here is the result of the preceding code:

```
C:\>java LambdaExpression1
Result of Addition: 5
Result of Division: 5
```

Summary

In this chapter, you learned about the concept of inheritance, interfaces, and inner classes. You also learned how to create a subclass and use the correct access specifiers while implementing inheritance. You came to know about calling superclass constructors and handling multilevel inheritance. You also explored in the chapter about handling overriding methods inherited from superclass and accessing overridden methods with the use of super keyword. Then, you studied about superclass variables with subclass objects, implementation of runtime polymorphism, how to create abstract classes, the use of final keyword and the lambda expression.

In the next chapter, you'll come across AWT package used for creating an applet.



6

AWT—Applets, Applications, and Event Handling

If you need an immediate solution to:	*See page:*
Using the Abstract Window Toolkit	175
Creating Applets	186
Using the <APPLET> HTML Tag	188
Handling Non-Java Browsers	189
Embedding <APPLET> Tags in Code	189
Using the init, start, stop, destroy, paint, and update Methods	190
Drawing Graphics in Applets	191
Reading Parameters in Applets	191
Using Java Consoles in Browsers	191
Adding Controls to Applets: Text Fields	192
Adding Controls to Applets: Buttons	193
Handling Events	194
Using Adapter Classes	200
Using Anonymous Inner Adapter Classes	202
Creating Windowed Applications	202
Exiting an Application When Its Window Is Closed	206
Applications You Can Run as Applets	206
Setting Applet Security Policies	207
Other Facilities in the java.awt Package	209

In Depth

We have worked through a lot of Java syntax to get to this point, and this is one of the payoff chapters. Here, we'll start working with graphical programs—both applets and applications. This chapter introduces the Java Abstract Windowing Toolkit (AWT), Java's original way of working with graphics. AWT is now supplemented with the Swing package, which you'll see in a few chapters. AWT provides the foundation of graphics work in Java, and even the Swing package is based on AWT.

In this chapter, we'll put AWT to work, creating applets and windowed applications. Before we begin, it's worth putting AWT in some historical perspective. AWT was developed very quickly for the first release of Java—in fact, in only six weeks. The original AWT developers used one window for each component of AWT, so each button, text field, checkbox, and so on has its own window as far as the underlying operating system is concerned. That turned out to be a considerable use of system resources as programs became larger. Sun has introduced the Swing package in which components are displayed using graphical methods of their containing applet or application windows—they don't have their own operating system windows. AWT components are called *heavyweight components* because of their significant use of system resources, and the Swing components are called *lightweight components* because they are just drawn and don't need their own windows. What does this mean for you? It's clear that, to Sun, Swing is the future. There are far more Swing components than AWT ones, and in fact, there's a Swing replacement component for each AWT component. Sun probably won't be expanding the AWT component set much in the future, whereas Swing is expected to grow. On the other hand, Swing itself is *based* on AWT—the windows that Swing uses to display Swing components (that is, windows, frame windows, applets, and dialog boxes) are all based on AWT containers. AWT isn't going away; and in order to work with Swing, you need a thorough background in AWT. For that reason and because so much development is still done with AWT—and more will be done in the future—we'll spend this and the next few chapters on AWT. We'll start with an overview of AWT itself.

The Abstract Windowing Toolkit

It's no exaggeration to say that the AWT was the driving force behind Java's popularity. You can create and display buttons, labels, menus, combo boxes, text fields, and the other user-interface controls you'd expect in windowed programs using AWT. Here's an overview of the most popular AWT classes:

- `Applet`—It creates an applet
- `Button`—It creates a button
- `Canvas`—It creates a canvas you can draw in
- `Checkbox`—It creates a checkbox
- `Choice`—It creates a choice control
- `Label`—It creates a label
- `Menu`—It creates a menu
- `ComboBox`—It creates a combo box
- `List`—It creates a list control
- `Frame`—It creates a frame for windowed applications
- `Dialog`—It creates a dialog box
- `Panel`—It creates a panel that can contain other controls
- `PopupMenu`—It creates a pop-up menu
- `RadioButton`—It creates a radio button
- `ScrollBar`—It creates a scrollbar
- `ScrollPane`—It creates a scrollable surface
- `TextArea`—It creates a two-dimensional text control
- `TextField`—It creates a one-dimensional text field (called a *text box* in other languages)

- `TextPane`—It creates a text surface
- `Window`—It creates a freestanding window

The AWT `Applet` class is what you base AWT applets on, and we'll take a look at that class first.

Applets

So, just what is an AWT applet? An applet is simply a Java class file that is embedded in a webpage, using HTML `<APPLET>` tag, to display graphics in a web brower. The browser automatically downloads the applet file and displays it in the appropriate space when the webpage gets loaded. In addition to displaying graphics, applets can display animation and handle various controls, such as text fields and buttons. Using applets makes your webpages *active*, not passive—that's their main attraction.

The process goes like this when working with AWT: You create a new applet, basing it on the `java.applet.Applet` class, which in turn is based on the AWT `Component` class. Here's an example which you have seen before, and we'll go through it again in this chapter. This example just displays the text "Kogent Learning Solutions!" in a web page:

```
import java.applet.Applet;
import java.awt.*;
public class AppletDemo extends Applet {
         public void paint(Graphics g) {
   g.drawString("Kogent Learning Solutions!", 100, 100); }
}
```

You compile the applet into a bytecode `.class` file. When you have your `.class` file, you upload it to an Internet Service Provider (ISP). You can give the applet the same protection you would give a web page, making sure anyone can read the `AppletDemo.class` file (for example, in Unix, you might give the applet the permission setting 644, which lets anyone read the file).

TIP

Unix file permissions make up three octal digits corresponding to the file owner's permission, the permission of others in the same user group, and the permission of all others. In each octal digit, a value of 4 indicates read permission, a value of 2 indicates write permission, and a value of 1 indicates execute permission. You add these values together to set the individual digits in a permission setting—for example, a permission of 0600 means that the file's owner, and only the file's owner, can both read and write to the file.

You can embed the new applet in a web page with the `<APPLET>` tag, indicating the name of the `.class` file for the applet as well as telling the web browser how much space (in pixels) to leave for the applet. Here's an example:

```
<HTML>
   <BODY>
         <CENTER>
                  <APPLET
                           CODE = "AppletDemo.class"
                           WIDTH = 300
                           HEIGHT = 200>
                  </APPLET>
         </CENTER>
   </BODY>
</HTML>
```

In this case, we have set up a centered 300-by-200-pixel space in a web page in which is to display the applet, and we told the web browser to download the `AppletDemo.class` file and run it. We'll cover the details, including the details on the `<APPLET>` tag, in this chapter. When the browser loads this page, it'll display the applet.

There's one important thing to know that we are working with applets—the Java web browser *plug-in*. Web browser manufacturers have been notoriously slow to implement the latest Java versions. This means that some of the Java applets we develop with Java 8 may *not* work in the latest browsers, even Microsoft Internet Explorer and Netscape Navigator (you can always use the Sun appletviewer, of course). We get dozens of letters asking why the examples in this book won't work—hadn't they tested? The fault, of course, is with the browsers, which

don't implement the latest Java version. Frustrated with this situation, Sun took the matter into its own hands by creating the Java Plug-In, which is a plug-in for Netscape Navigator and an ActiveX control for Microsoft Internet Explorer. The plug-in is a complete implementation of the Java Runtime Environment and lets you run applets using the latest Java version. Note that you must take some special steps to configure your web pages to use the plug-in, but Sun also creates and distributes a utility that we'll use here, the HTML Converter, to automatically convert any web page to use the plug-in.

Applications

AWT-windowed applications are based on the AWT `Frame` class, which creates a window with a frame that displays buttons and a title. Here's an example that you'll see more of in this chapter. Like the previous applet, this application displays "Kogent Learning Solutions!" in a frame window:

```
import java.awt.*;
import java.awt.event.*;
public class ApplicationDemo extends Frame {
        public void paint(Graphics g) {
  g.drawString("Kogent Learning Solutions!", 50, 100); }
  public ApplicationDemo() { addWindowListener(new MyWindowAdapter()); }
  public static void main(String [] args) {
        ApplicationDemo demo = new ApplicationDemo();
        demo.setSize(250,250);
        demo.setVisible(true); }
}
class MyWindowAdapter extends WindowAdapter {
  public void windowClosing(WindowEvent e) { System.exit(0); }
}
```

After compiling and executing the above program "`ApplicationDemo.java`" file, we'll get the output written "Kogent Learning Solutions!" in a frame window having width and height both 250. We'll go through the process of creating windowed applications in detail in this chapter.

Handling Events

One of the biggest aspects of creating applets and applications is letting the user interact with the program, and you do that with *events*. When the user performs some action—clicking a button, closing a window, selecting an item in a list, or using the mouse, for example—Java considers that as an event. We'll handle events throughout the rest of this book, and we'll take a look at how event handling works in this chapter.

To start working with events, we'll also introduce two basic controls in this chapter—buttons and text fields. The user can use mouse to click buttons that initiate some action in your program, such as placing some text in a text field. In fact, the button click is perhaps the most basic event Java supports. To illustrate how event handling works in Java, we'll create programs in this chapter which support buttons and text fields. The full details on these controls appear in the next chapter.

Now that you have an overview of applets, applications, and event handling, it's time to get to the details in the "Immediate Solutions" section.

Immediate Solutions

Using the Abstract Window Toolkit

"OK," says the Novice Programmer. "I'm ready to start working with the AWT. Where does it start?" "With the `Component` class," you say. "Get some coffee and we'll take a look at it."

The most basic AWT class is the `java.awt.Component` class, which all AWT visual components are based on. For example, the AWT `Button` class, `java.awt.Button`, is derived directly from `java.awt.Component`. The `java.awt.Component` class itself is derived directly from the `java.lang.Object` class, which you saw in the previous chapter.

The `Component` class includes a large number of methods, many of which you'll see in this and the upcoming chapter. We have listed them all in Table 6.1 for reference. This is a long table, but it's worth glancing it through and coming back later for reference.

Table 6.1: Methods of the AWT Component class

Method	Does this
`boolean action (Event evt, Object what)`	It is deprecated. You should register the component as an `ActionListener` on component which fires action events
`void add(PopupMenu popup)`	It adds the pop-up menu to the component with an `ActionListener`
`void addComponentListener (ComponentListener l)`	It adds the component listener to receive component events
`void addFocusListener(FocusListener l)`	It adds the focus listener to receive focus events
`void addHierarchyBoundsListener (HierarchyBoundsListener l)`	It adds the hierarchy bounds listener to receive hierarchy bounds events from this component when the hierarchy to which this container belongs changes
`void addHierarchyListener (HierarchyListener l)`	It adds the hierarchy listener to receive hierarchy changed events from this component when the hierarchy to which this container belongs changes
`void addInputMethodListener (InputMethodListener l)`	It adds the input method listener to receive input method events
`void addKeyListener (KeyListener l)`	It adds the key listener to receive key events
`void addMouseListener (MouseListener l)`	It adds the mouse listener to receive mouse events
`void addMouseMotionListener (MouseMotionListener l)`	It adds the mouse motion listener to receive mouse motion events
`void addMouseWheelListener (MouseWheelListener I)`	It adds the mouse wheel listener to receive mouse wheel events
`void addNotify()`	It makes a component "displayable" by connecting it to a native screen resource
`void addPropertyChangeListener (PropertyChangeListener listener)`	It adds a `PropertyChangeListener`
`void addPropertyChangeListener (String propertyName, PropertyChangeListener listener)`	It adds a `PropertyChangeListener` for specified property
`void applyComponentOrientation (ComponentOrientation orientation)`	It sets the `ComponentOrientation` property of this component and all components contained within it
`boolean areFocusTraversalKeysSet (int id)`	It returns whether the focus traversal keys are explicitly defined
`Rectangle bounds()`	It is deprecated and replaced by `getBounds()`

Table 6.1: Methods of the AWT Component class

Method	Does this
`int checkImage(Image image, ImageObserver observer)`	It gets the status of the screen representation of the specified image
`int checkImage (Image image, int width, int height ImageObserver observer)`	It gets the status of the screen representation of the specified image
`protected AWTEvent coalesceEvents (AWTEvent existingEvent, AWTEvent newEvent)`	It coalesces an event being posted with an existing event
`boolean contains (int x, int y)`	It checks whether this component contains the indicated point
`boolean contains(Point p)`	It checks whether this component contains the indicated point
`Image createImage (ImageProducer producer)`	It creates an image from the indicated image producer
`Image createImage (int width, int height)`	It creates an offscreen drawable image to be used for double buffering
`VolatileImage createVolatileImage (int width, Int height)`	It creates a volatile-off screen drawable image
`VolatileImage createVolatileImage(int width, Int height, Image Capabilities caps)`	It creates a volatile off-screen drawable image with the given capabilities
`void deliverEvent(Event e)`	It is deprecated and replaced by `dispatchEvent(AWTEvent)`
`void disable()`	It is deprecated and replaced by `setEnabled(boolean)`
`protected void disableEvents (long eventsToDisable)`	It disables the events defined by the indicated event mask parameter from being sent to this component
`void dispatchEvent (AWTEvent e)`	It dispatches an event to this component or one of its subcomponents
`void doLayout()`	It makes the layout manager lay out this component
`void enable()`	It is deprecated and replaced by `setEnabled(boolean)`
`void enable(boolean b)`	It is deprecated and replaced by `setEnabled(boolean)`
`protected void enableEvents (long eventsToEnable)`	It enables the events defined by the indicated event mask parameter to be sent to this component
`void enableInputMethods (boolean enable)`	It enables or disables input method support
`protected void firePropertyChange (String propertyName,boolean oldValue, boolean newValue)`	It sets up support for reporting bound property changes for boolean properties
`void firePropertyChange (String propertyName, byte oldValue, byte newValue)`	It reports a bound property change
`void firePropertyChange (String propertyName, char oldValue, char newValue)`	It reports a bound property change
`void firePropertyChange (String propertyName, double oldValue, double newValue)`	It reports a bound property change
`void firePropertyChange (String propertyName, float oldValue, float newValue)`	It reports a bound property change
`protected void firePropertyChange (String propertyName, int oldValue, int newValue)`	It supports for reporting bound property changes for integer property

Table 6.1: Methods of the AWT Component class

Method	Does this
`void firePropertyChange (String propertyName, long oldValue, long newValue)`	It reports a bound property change
`void firePropertyChange (String propertyName, Object oldValue, Object newValue)`	It supports for reporting bound property changes for Object property
`void firePropertyChange (String propertyName, short oldValue, short newValue)`	It reports a bound property change
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this component
`float getAlignmentX()`	It gets the alignment along the x-axis
`float getAlignmentY()`	It gets the alignment along the y-axis
`Color getBackground()`	It gets the background color of this component
`int getBaseline(int width, int height)`	It returns the baseline
`Component.BaselineResizeBehavior getBaselineResizeBehavior()`	It returns an enum specifying how the component baseline changes with change in size
`Rectangle getBounds()`	It gets the bounds of this component in a `Rectangle` object
`Rectangle getBounds(Rectangle rv)`	It stores the bounds of this component into "return value" `rv` and `return rv`
`ColorModel getColorModel()`	It gets the instance of `ColorModel` used to display the component
`Component getComponentAt (int x, int y)`	It determines whether this component or one of its immediate subcomponents contains the (x, y) location and gets the containing component
`Component getComponentAt(Point p)`	It gets the component or subcomponent that contains the indicated point
`ComponentListener[] getComponentListeners()`	It returns an array of all the component listeners registered with this component
`ComponentOrientation getComponentOrientation()`	It retrieves the language-sensitive orientation to be used to order the elements or text within this component
`Cursor getCursor()`	It gets the cursor set in the component
`DropTarget getDropTarget()`	It gets the `DropTarget` connected to this component
`Container getFocusCycleRootAncestor()`	It returns the container which is the focus cycle root of the component
`Font getFont()`	It gets the font of this component
`FontMetrics getFontMetrics (Font font)`	It gets the font metrics for the indicated font
`Color getForeground()`	It gets the foreground color of this component
`Graphics getGraphics()`	It creates a graphics context for this component
`GraphicsConfiguration getGraphicsConfiguration()`	It gets the `GraphicsConfiguration` associated with this component
`int getHeight()`	It returns the current height of this component
`HierarchyBoundsListener[] getHierarchyBoundsListeners()`	It returns an array of all the hierarchy bound listeners registered with this component
`HierarchyListener[] getHierarchyListeners()`	It returns an array of all the hierarchy listeners registered with this component

Table 6.1: Methods of the AWT Component class

Method	Does this
`boolean getIgnoreRepaint()`	It returns True when messages received from the operating system should be painted; otherwise, it returns False
`InputContext getInputContext()`	It gets the input context used by this component
`InputMethodListener[] getInputMethodListeners()`	It return an array of all input method listeners registered with this component
`InputMethodRequests getInputMethodRequests()`	It gets the input method request handler that supports requests from input methods for this component
`KeyListener[] getKeyListeners()`	It return an array of all key listeners registered with this component
`<T extends EventListener> T[] getListeners(Class<T> listenerType)`	It returns an array of all objects that are currently registered as `FooListeners`
`Locale getLocale()`	It gets the locale of this component
`Point getLocation()`	It gets the location of this component in the form of a point specifying the component's top-left corner
`Point getLocation (Point rv)`	It stores the x,y origin of this component into "return value" `rv` and return `rv`
`Point getLocationOnScreen()`	It gets the location of this component in the form of a point specifying the component's top-left corner
`Dimension getMaximumSize()`	It gets the maximum size of this component
`Dimension getMinimumSize()`	It gets the minimum size of this component
`MouseListener[] getMouseListeners()`	It returns an array of all the mouse listeners registered with this component
`MouseMotionListener[] getMouseMotionListeners()`	It returns an array of all the mouse motion listeners registered with this component
`Point getMousePosition()`	It returns the position of the mouse pointer
`MouseWheelListener[] getMouseWheelListeners()`	It returns an array of all the mouse wheel listeners registered with this component
`String getName()`	It gets the name of the component
`Container getParent()`	It gets the parent of this component
`java.awt.peer.ComponentPeer getPeer()`	It is deprecated and replaced by `boolean isDisplayable()`
`Dimension getPreferredSize()`	It gets the preferred size of this component
`PropertyChangeListener[] getPropertyChangeListeners()`	It returns an array of all the property change listeners registered with this component
`PropertyChangeListener[] getPropertyChangeListeners (String propertyName)`	It returns an array of all the listeners which have been associated with the named property
`Dimension getSize()`	It gets the size of this component in a `Dimension` object
`Dimension getSize(Dimension rv)`	It stores the width/height of this component into "return value" `rv` and return `rv`
`Toolkit getToolkit()`	It gets the toolkit of this component
`Object getTreeLock()`	It gets the locking object for the AWT component-tree and layout operations
`int getWidth()`	It gets the current width of this component
`int getX()`	It gets the current x coordinate of the component's origin
`int getY()`	It gets the current y coordinate of the component's origin

Table 6.1: Methods of the AWT Component class

Method	Does this
`boolean gotFocus (Event evt, Object what)`	It is deprecated and replaced by `processFocusEvent(FocusEvent)`
`boolean handleEvent (Event evt)`	It is deprecated and replaced by `processEvent(AWTEvent)`
`boolean hasFocus()`	It returns true if this component has the focus
`void hide()`	It is deprecated and replaced by `setVisible(boolean)`
`boolean imageUpdate(Image img, int flags, int x, int y, int w, int h)`	It repaints the component when the image has changed
`boolean inside (int x, int y)`	It is deprecated and replaced by `contains(int, int)`
`void invalidate()`	It invalidates the component
`boolean isBackgroundSet()`	It returns True when the background color has been explicitly set; otherwise, it returns False
`boolean isCursorSet()`	It returns True when the cursor has been explicitly set; otherwise, it returns False
`boolean isDisplayable()`	It determines whether the component can be displayed
`boolean isDoubleBuffered()`	It returns True if this component is painted to an offscreen image that's copied to the screen later
`boolean isEnabled()`	It indicates whether this component is enabled
`boolean isFocusable()`	It indicates whether this component can be focused
`boolean isFocusTraversable()`	It indicates whether this component can be traversed using Tab or Shift+Tab
`boolean isFocusCycleRoot (Container container)`	It shows whether the specified container is the focus cycle root of the component's focus traversal cycle
`boolean isFocusOwner()`	It returns true if this component is the focus owner
`boolean isFontSet()`	It returns True when the font set for this component has been explicitly set; otherwise, it returns False
`boolean isForegroundSet()`	It returns True when the foreground color for this component has been explicitly set; otherwise, it returns False
`boolean isLightweight()`	It indicates whether the component is lightweight (a lightweight component doesn't have a native toolkit peer)
`boolean isMaximumSizeSet()`	It returns True when the maximum size has been set to a non-null value; otherwise, it returns False
`boolean isMinimumSizeSet()`	It returns True when the minimum size has been set to a non-null value; otherwise, it returns False
`boolean isOpaque()`	It returns True if this component is completely opaque; False by default
`boolean isPreferredSizeSet()`	It returns True when the preferred size has been set to a non-null value; otherwise, it returns False
`boolean isShowing()`	It determines whether this component is visible onscreen
`boolean isValid()`	It determines whether this component is valid
`boolean isVisible()`	It determines whether this component should be visible when its parent is visible
`boolean keyDown (Event evt, int key)`	It is deprecated and replaced by `processKeyEvent(KeyEvent)`

Table 6.1: Methods of the AWT Component class

Method	Does this
`boolean keyUp (Event evt, int key)`	It is deprecated and replaced by `processKeyEvent(KeyEvent)`
`void layout()`	It is deprecated and replaced by `doLayout()`
`void list()`	It prints a listing of this component to the standard output stream `System.out`
`void list(PrintStream out)`	It prints a listing of this component to the indicated output stream
`void list (PrintStream out, int indent)`	It prints out a list, starting at the indicated indention, to the indicated print stream
`void list(PrintWriter out)`	It prints a listing to the indicated print writer
`void list(PrintWriter out, int indent)`	It prints out a list, starting at the indicated indention, to the indicated print writer
`Component locate (int x, int y)`	It is deprecated and replaced by `getComponentAt(int, int)`
`Point location()`	It is deprecated and replaced by `getLocation()`
`boolean lostFocus (Event evt, Object what)`	It is deprecated and replaced by `processFocusEvent(FocusEvent)`
`Dimension minimumSize()`	It is deprecated and replaced by `getMinimumSize()`
`boolean mouseDown (Event evt, int x, int y)`	It is deprecated and replaced by `processMouseEvent(MouseEvent)`
`boolean mouseDrag (Event evt, int x, int y)`	It is deprecated and replaced by `processMouseMotionEvent(MouseEvent)`
`boolean mouseEnter (Event evt, int x, int y)`	It is deprecated and replaced by `processMouseEvent(MouseEvent)`
`boolean mouseExit (Event evt, int x, int y)`	It is deprecated and replaced by `processMouseEvent(MouseEvent)`
`boolean mouseMove (Event evt, int x, int y)`	It is deprecated and replaced by `processMouseMotionEvent(MouseEvent)`
`boolean mouseUp (Event evt, int x, int y)`	It is deprecated and replaced by `processMouseEvent(MouseEvent)`
`void move(int x, int y)`	It is deprecated and replaced by `setLocation(int, int)`
`void nextFocus()`	It is deprecated and replaced by `transferFocus()`
`void paint(Graphics g)`	It paints this component
`void paintAll(Graphics g)`	It paints this component and all subcomponents
`protected String paramString()`	It gets a string representing the state of this component
`boolean postEvent(Event e)`	It is deprecated and replaced by `dispatchEvent(AWTEvent)`
`Dimension preferredSize()`	It is deprecated and replaced by `getPreferredSize()`
`boolean prepareImage (Image image,ImageObserver observer)`	It prepares an image for rendering on this component
`boolean prepareImage(Image image, int width, int height, ImageObserver observer)`	It prepares an image for rendering on this component at the indicated width and height
`void print(Graphics g)`	It prints the component
`void printAll(Graphics g)`	It prints the component and all its subcomponents

Table 6.1: Methods of the AWT Component class

Method	Does this
`protected void processComponentEvent (ComponentEvent e)`	It processes the component events occurring in this component by sending them to any registered `ComponentListener` objects
`protected void processEvent(AWTEvent e)`	It processes events occurring on this component
`protected void processFocusEvent(FocusEvent e)`	It processes focus events occurring in this component by dispatching them to any registered `FocusListener` objects
`protected void processHierarchyBoundsEvent(HierarchyEvent e)`	It processes hierarchy bounds events occurring on this component by dispatching them to any registered `HierarchyBoundsListener` objects
`protected void processHierarchyEvent (HierarchyEvent e)`	It processes hierarchy events occurring on this component by dispatching them to any registered `HierarchyListener` objects
`protected void processInputMethodEvent (InputMethodEvent e)`	It processes input method events occurring on this component by sending them to any registered `InputMethodListener` objects
`protected void processKeyEvent(KeyEvent e)`	It processes key events occurring on this component by sending them to any registered `KeyListener` objects
`protected void processMouseEvent (MouseEvent e)`	It processes mouse events occurring on this component by sending them to any registered `MouseListener` objects
`protected void processMouseMotionEvent (MouseEvent e)`	It processes mouse motion events occurring on this component by sending them to any registered `MouseMotionListener` objects
`protected void processMouseWheelEvent (MouseEvent e)`	It processes mouse wheel events occurring on this component by sending them to any registered `MouseWheelListener` objects
`void remove (MenuComponent popup)`	It removes the indicated pop-up menu from the component
`void removeComponentListener (ComponentListener l)`	It removes the indicated component listener so that it no longer receives component events
`void removeFocusListener (FocusListener l)`	It removes the indicated focus listener so that it no longer receives focus events
`void removeHierarchyBoundsListener(HierarchyBoundsListener l)`	It removes the indicated hierarchy bounds listener so that it no longer receives hierarchy bounds events
`void removeHierarchyListener (HierarchyListener l)`	It removes the indicated hierarchy listener so that it no longer receives hierarchy changed events
`void removeInputMethodListener (InputMethodListener l)`	It removes the indicated input method listener so that it no longer receives input method events
`void removeKeyListener (KeyListener l)`	It removes the indicated key listener so that it no longer receives key events
`void removeMouseListener (MouseListener l)`	It removes the indicated mouse listener so that it no longer receives mouse events
`void removeMouseMotionListener (MouseMotionListener l)`	It removes the indicated mouse motion listener so that it no longer receives mouse motion events
`void removeMouseWheelListener (MouseMotionListener l)`	It removes the indicated mouse wheel listener so that it no longer receives mouse wheel events
`void removeNotify()`	It disallows displaying the component by destroying its native screen resource

Table 6.1: Methods of the AWT Component class

Method	Does this
`void removePropertyChangeListener(PropertyChangeListener listener)`	It removes a `PropertyChangeListener`
`void removePropertyChangeListener(String propertyName, PropertyChangeListener listener)`	It removes a `PropertyChangeListener` for the given property
`void repaint()`	It repaints the component
`void repaint(int x, int y, int width, int height)`	It repaints the indicated rectangle of this component
`void repaint(long tm)`	It repaints the component
`void repaint(long tm, int x, int y, int width, height)`	It repaints the indicated rectangle of this `int` component within `tm` milliseconds
`void requestFocus()`	It requests the input focus
`protected void requestFocus(boolean temporary)`	It requests the input focus, and the component's top-level ancestor becomes the focused `Window`
`boolean requestFocusInWindow()`	It requests the input focus if the top-level ancestor is already the focused `Window`
`protected boolean requestFocusInWindow(boolean temporary)`	It requests the input focus if the top-level ancestor is already the focused `Window`
`void reshape(int x, int y, int width, int height)`	It is deprecated and replaced by `setBounds(int, int, int, int)`
`void resize(Dimension d)`	It is deprecated and replaced by `setSize(Dimension)`
`void resize(int width, int height)`	It is deprecated and replaced by `setSize(int, int)`
`void revalidate()`	It revalidates the component hierarchy up to the nearest validate root
`void setBackground (Color c)`	It sets the background color
`void setBounds(int x, int y, int width, int height)`	It moves and resizes the component
`void setBounds (Rectangle r)`	It moves and resizes the component to conform to the new bounding rectangle `r`
`void setComponentOrientation (ComponentOrientation o)`	It sets the language-sensitive orientation to be used to order the elements or text
`void setCursor (Cursor cursor)`	It sets the cursor image to the indicated cursor
`void setDropTarget (DropTarget dt)`	It associates a `DropTarget` with this component
`void setEnabled(boolean b)`	It enables or disables this component, depending on the value of the parameter `b`
`void setFocusable (boolean focusable)`	It sets the focusable state of the component
`void setFocusTraversalKeys (int id, Set<? extends AWTKeyStroke> keyStrokes)`	It sets the focus traversal keys for a given traversal operation
`void setFocusTraversalKeysEnabled(boolean focusTraversalKeysEnabled)`	It sets whether the traversal keys are enabled for the component
`void setFont(Font f)`	It sets the font of the component

Table 6.1: Methods of the AWT Component class	
Method	**Does this**
`void setForeground (Color c)`	It sets the foreground color of the component
`void setIgnoreRepaint (boolean ignoreRepaint)`	It sets whether or not paint messages received from the operating system
`void setLocale(Locale l)`	It sets the locale of the component
`void setLocation (int x, int y)`	It moves this component to a new location
`void setLocation(Point p)`	It moves this component to a new location
`void setMaximumSize(Dimension maximumSize)`	It sets the maximum size of the component to a constant value
`void setMinimumSize(Dimension minimumSize)`	It sets the minimum size of the component to a constant value
`void setName(String name)`	It sets the name of the component to the indicated string
`void setPreferredSize(Dimension Preferred size)`	It sets the preferred size of the component to a constant value
`void setSize(Dimension d)`	It resizes this component so that it has width `d.width` and height `d.height`
`void setSize (int width, int height)`	It resizes this component so that it has width `width` and height `height`
`void setVisible(boolean b)`	It shows or hides this component as specified by the value of parameter `b`
`void show()`	It is deprecated and replaced by `setVisible(boolean)`
`void show(boolean b)`	It is deprecated and replaced by `setVisible(boolean)`
`Dimension size()`	It is deprecated and replaced by `getSize()`
`String toString()`	It gets a string representation of the component
`void transferFocus()`	It transfers the focus to the next component
`void transferFocusBackward()`	It transfers the focus to the previous component
`void transferFocusUpCycle()`	It transfers the focus up one focus traversal cycle
`void update(Graphics g)`	It updates the component
`void validate()`	It ensures that the component has a valid layout in effect

Another important AWT class is the `Container` class. This class is derived from AWT `Component` class, and it's the basis of AWT containers, which can hold other components. Applets and windowed applications, which can display AWT components, are based on the `Container` class. Because you run across the `Container` class often in AWT programming—as in the next topic coming up—we have listed its methods in Table 6.2:

Table 6.2: Methods of the Container class	
Method	**Does this**
`Component add(Component comp)`	It adds the indicated component to this container
`Component add(Component comp, int index)`	It adds the indicated component to this container at a specific position
`void add(Component comp, Object constraints)`	It adds the indicated component to this container
`void add(Component comp, Object constraints, int index)`	It adds the indicated component to this container with the indicated constraints at the indicated index
`Component add(String name, Component comp)`	It adds the indicated component to this container

Table 6.2: Methods of the Container class

Method	Does this
`void addContainerListener(ContainerListener l)`	It adds the indicated container listener to receive container events from the container
`protected void addImpl (Component comp Object constraints, int index)`	It adds the indicated component to the container at the given index
`void addPropertyChangeListener(PropertyChangeListener listener)`	It adds a `PropertyChangeListener`
`void addPropertyChangeListener(String propertyName, PropertyChangeListener listener)`	It adds a `PropertyChangeListener` for the indicated property
`void applyComponentOrientation(ComponentOrientation o)`	It sets the `ComponentOrientation` property of this container and all components contained inside it
`boolean areFocusTraversalKeysSet (int id)`	It returns True when the set of focus traversal keys for the given focus traversal operation are explicitly defined; otherwise, it returns False
`int countComponents()`	It is deprecated and replaced by `getComponentCount()`
`void deliverEvent(Event e)`	It is deprecated and replaced by `dispatchEvent(AWTEvent)`
`void doLayout()`	It makes the container lay out its components
`Component findComponentAt (int x, int y)`	It locates the visible child component containing the indicated position
`Component findComponentAt (Point p)`	It locates the visible child component containing the indicated point
`float getAlignmentX()`	It gets the alignment along the x axis
`float getAlignmentY()`	It gets the alignment along the y axis
`Component getComponent (int n)`	It gets the nth component in this container
`Component getComponentAt (int x, int y)`	It locates the component that contains the x,y position
`Component getComponentAt (Point p)`	It gets the component that contains the indicated point
`int getComponentCount()`	It gets the number of components in this panel
`Component[] getComponents()`	It gets all the components in the container
`int getComponentZOrder (Component comp)`	It returns the z-order index of the component within the container
`ContainerListener[] getContainerListeners()`	It returns an arrays of all container listeners registered with this container
`Set<AWTKeyStroke> getFocusTraversalKeys(int id)`	It returns the set of focus traversal keys for a specific traversal operation
`FocusTraversalPolicy getFocusTraversalPolicy()`	It returns the focus traversal policy or null if the container is not a focus cycle root
`Insets getInsets()`	It determines the insets of the container, which indicate the size of the container's border
`LayoutManager getLayout()`	It gets the layout manager for the container
`<T extends EventListener>T[] getListeners(Class<T> listenerType)`	It returns an array of all listeners currently registered as `FooListeners`
`Dimension getMaximumSize()`	It gets the maximum size of the container

Table 6.2: Methods of the Container class

Method	Does this
`Dimension getMinimumSize()`	It gets the minimum size of the container
`Point getMousePosition(Boolean allowChildren)`	It returns the position of the mouse pointer if the container is under the mouse pointer; otherwise, it returns null
`Dimension getPreferredSize()`	It gets the preferred size of the container
`Insets insets()`	It is deprecated and replaced by `getInsets()`
`void invalidate()`	It invalidates the container
`boolean isAncestorOf(Component c)`	It checks whether the component is contained in the hierarchy of this container
`boolean isFocusCycleRoot()`	It returns True when the container is the root of a focus traversal cycle; otherwise, it returns False
`boolean isFocusCycleRoot(Container container)`	It returns True when the indicated container is the root of a focus traversal cycle; otherwise, it returns False
`boolean isFocusTraversalPolicySet(Container container)`	It returns True when the focus traversal policy has been explicitly set for the container; otherwise, it returns False
`boolean isValidateRoot()`	It checks whether the container is a validate root
`void layout()`	It is deprecated and replaced by `doLayout()`
`void list(PrintStream out, int indent)`	It prints a listing of this container to the indicated output stream
`void list(PrintWriter out, int indent)`	It prints out a list, starting at the indicated indention, to the indicated print writer
`Component locate (int x, int y)`	It is deprecated and replaced by `getComponentAt(int, int)`
`Dimension minimumSize()`	It is deprecated and replaced by `getMinimumSize()`
`void paint(Graphics g)`	It paints the container
`void paintComponents(Graphics g)`	It paints each component in the container
`protected String paramString()`	It gets the parameter string representing the state of the container
`Dimension preferredSize()`	It is deprecated and replaced by `getPreferredSize()`
`void print(Graphics g)`	It prints the container
`void printComponents(Graphics g)`	It prints each of the components in this container
`protected void processContainerEvent (ContainerEvent e)`	It processes container events occurring on this container by sending them to `ContainerListener` objects
`protected void processEvent (AWTEvent e)`	It processes events in this container
`void remove(Component comp)`	It removes the indicated component from the container
`void remove(int index)`	It removes the component at the index from the container
`void removeAll()`	It removes all the components from this container
`void removeContainerListener (ContainerListener l)`	It removes the indicated container listener so that it no longer receives container events from this container
`void removeNotify()`	It makes this container undisplayable by removing its connection to its native screen resource
`void setComponentZOrder (Component comp, int index)`	It moves the indicated component to the indicated z-order index in the container

Table 6.2: Methods of the Container class	
Method	**Does this**
`void setFocusCycleRoot(boolean focusCycleRoot)`	It sets whether the container is the root of a focus traversal cycle
`void setFocusTravsersalKeys(int id, <Set? extends AWTKeyStroke> keystrokes)`	It sets the focus traversal keys for a specific traversal operation
`void setFocusTraversalPolicy(FocusTraversalPolicy policy)`	It sets the focus traversal policy if the container is a focus cycle root
`void setFocusTraversalPolicyProvider(boolean provider)`	It sets whether the container will be used to provide the focus traversal policy
`void setFont(Font f)`	It sets the font of the container
`void setLayout (LayoutManager mgr)`	It sets the layout manager for the container
`void transferFocusDownCycle()`	It transfers the focus down one focus traversal cycle
`void update(Graphics g)`	It updates the container
`void validate()`	It validates this container and all its subcomponents
`protected void validateTree()`	It recursively descends the container tree and recomputes the layout for any subtrees marked as needing it (those marked as invalid)

Creating Applets

"Finally!" says the Novice Programmer. "At last, I'm ready to create an applet." "That's right," you say. "Now which one would you like to create?" "Hmm," says the NP.

You base your applets on the `java.applet.Applet` class, which is a subclass of the `java.awt.Panel` class:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |_____java.awt.Panel
                    |____java.applet.Applet
```

For the sake of reference, you'll find the methods of the `Applet` class in Table 6.3.

Table 6.3: Methods of the Applet class	
Method	**Does this**
`void destroy()`	It is called when the applet is about to be disposed of
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this applet
`AppletContext getAppletContext()`	It determines this applet's context to allow the applet query and affect the environment in which it runs
`String getAppletInfo()`	It gets information about this applet
`AudioClip getAudioClip (URL url)`	It gets the `AudioClip` object specified by the `URL` argument
`AudioClip getAudioClip (URL url, String name)`	It gets the `AudioClip` object specified by the `URL` and `name` arguments
`URL getCodeBase()`	It gets the base URL
`URL getDocumentBase()`	It gets the document URL
`Image getImage(URL url)`	It gets an image that can then be painted on the screen
`Image getImage (URL url, String name)`	It gets an image that can then be painted on the screen

Table 6.3: Methods of the Applet class

Method	Does this
`Locale getLocale()`	It gets the locale for the applet
`String getParameter (String name)`	It gets the value of the named parameter in the HTML tag
`String[][] getParameterInfo()`	It gets information about the applet parameters
`void init()`	It is called by the browser or appletviewer to allow you to initialize the applet
`boolean isActive()`	It determines whether this applet is active
`boolean isValidateRoot()`	It determines whether this container is a validate root
`static AudioClip newAudioClip(URL url)`	It gets an audio clip from the given URL
`void play(URL url)`	It plays the audio clip at the specified absolute URL
`void play(URL url, String name)`	It plays the audio clip given the URL and a specifier that's relative to it
`void resize(Dimension d)`	It requests that this applet be resized
`void resize (int width, int height)`	It requests that this applet be resized
`void setStub(AppletStub stub)`	It sets this applet's stub
`void showStatus(String msg)`	It requests that the string be displayed in the status window of the applet
`void start()`	It is called by the browser or applet viewer to tell the applet it should start execution
`void stop()`	It is called by the browser or applet viewer to tell the applet it should stop execution

Let's look at an example. Here, we'll create an applet that displays the text "Kogent Learning Solutions!" in the file `CreateApplet.java`. We start by deriving a new class `CreateApplet` from the `java.applet.Applet` class, like this:

```
import java.applet.Applet;
public class CreateApplet extends Applet { . . . }
```

To display the message in the applet, we'll use the `paint()` method, which the applet inherits from the `Component` class. When an applet is displayed, its `paint()` method is called, and you can place the code for drawing the applet in that method. The `paint()` method is passed an object of the `Graphics` class, which you'll study in a few chapters. It's the basis of graphical work in applets. This object supports a method named `drawString()`, which we'll use to draw a string of text in the applet. The `Graphics` class is an AWT class, so we'll import AWT classes when we override the default `paint()` method of the `Applet` class like this:

```
import java.applet.Applet;
import java.awt.*;
public class CreateApplet extends Applet {
        public void paint(Graphics g) { . . . }
}
```

Now we'll customize the overridden `paint()` method to draw the text "Kogent Learning Solutions!" at location (80, 100) in the applet. Applet coordinates are in pixels; therefore, (80, 100) is at 80 pixels from the left edge of the applet and 100 pixels here the bottom of the title bar. Here's the code:

```
import java.applet.Applet;
import java.awt.*;
/*<APPLET
  CODE= CreateApplet.class
  WIDTH=300
  HEIGHT=200 >
</APPLET>*/
public class CreateApplet extends Applet {
        public void paint(Graphics g) {
```

```
    g.drawString("Kogent Learning Solutions!", 80, 100); }
    }
```

This new `paint()` method draws directly on the surface of the applet. Note that you can also place controls such as buttons and text fields directly on the surface of the applet. You'll see how that works in this chapter as well. Once we compile `CreateApplet.java` into `CreateApplet.class` using `javac`, the applet is created. Now it's time to see the applet at work.

Using the <APPLET> HTML Tag

"OK," says the Novice Programmer. "I have created an applet and compiled it into a class file. How do I actually *look* at it?" "You have to use the HTML `<APPLET>` tag," you reply.

When you have created an applet's `.class` file, you can upload the applet to an ISP (or view it on your own machine) and set its protection (see the introduction to this chapter) so that it may be read. Next, you can create a web page using the `<APPLET>` tag to display the applet. Here's what the `<APPLET>` tag looks like:

```
    <APPLET
          [CODEBASE = URL]
          CODE = filename
          [ALT = alternateText]
          [NAME = instancename]
          WIDTH = pixels
          HEIGHT = pixels
          [ALIGN = alignment]
          [VSPACE = pixels]
          [HSPACE = pixels]
    > [<PARAM NAME = name VALUE = value>]
          . . .
          [<PARAM = NAME name VALUE = value>]
</APPLET>
```

Here are the attributes of the `<APPLET>` tag:

- `CODEBASE`—The URL that specifies the directory in which the applet code is to be found.
- `CODE`—The name of the applet file, including the extension `.class`.
- `ALT`—The text to be displayed if a browser supports applets but cannot run this one for some reason.
- `NAME`—The name of the applet in the web browser. Applets must be given names if you want other applets to be able to find and interact with them.
- `WIDTH`—The width of the space reserved for the applet.
- `HEIGHT`—The height of the space reserved for the applet.
- `ALIGN`—The alignment of the applet: `LEFT`, `RIGHT`, `TOP`, `BOTTOM`, `MIDDLE`, `BASELINE`, `TEXTTOP`, `ABSMIDDLE`, or `ABSBOTTOM`.
- `VSPACE`—The space allocated above and below the applet.
- `HSPACE`—The space allocated to the right and left around the applet.
- `PARAM NAME`—The name of a parameter to pass to the applet.
- `PARAM VALUE`—The value of a parameter.

Here's a web page called `Applet.html` that displays the applet created in the previous topic (note that the only required attributes are the `CODE`, `WIDTH`, and `HEIGHT` attributes):

```
<HTML>
   <BODY>
         <CENTER>
               <APPLET
                     CODE=CreateApplet.class
                     WIDTH=300
                     HEIGHT=200 >
               </APPLET>
         </CENTER>
   </BODY>
</HTML>
```

In this case, we are not specifying a code base, so we place `CreateApplet.class` in the same directory as `Applet.html`. This web page, as opened in Internet Explorer, appears in Figure 6.1. Note that if an applet doesn't work in your browser, it could be well because the browser doesn't support the current Java version.

You can also use the Sun appletviewer, which comes with Java, to view the applet. You open the web page like this (this example assumes that Applet.html is in the current directory—you can always give the path of the `.html` file like this: appletviewer `C:\BB_Chapters\Chapter-6\Applet.html`):

```
C:\>appletviewer Applet.html
```

The result appears in Figure 6.2. The appletviewer always supports the latest version of Java, so if your web browser doesn't, and you don't want to install the Java plug-in, you can always use the appletviewer to test your applets.

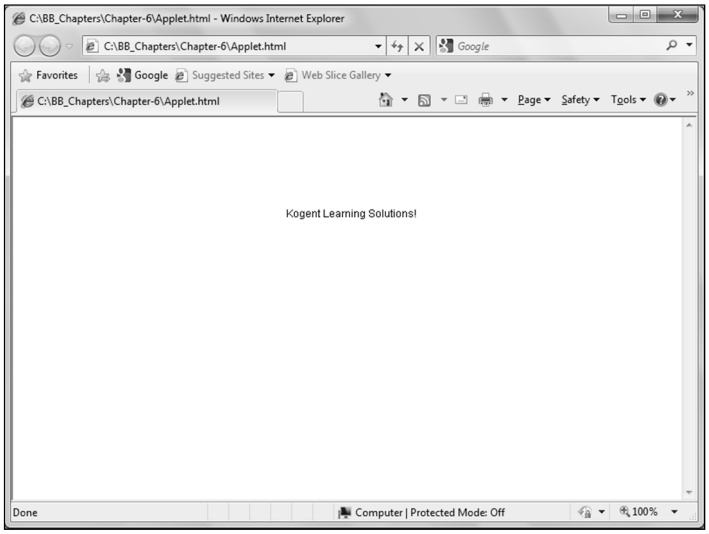


Figure 6.1: An applet at work in the Internet Explorer

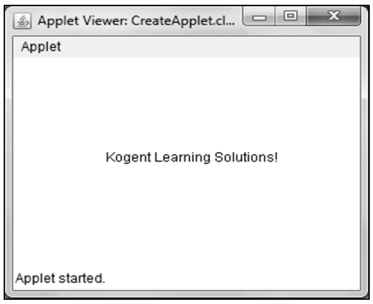


Figure 6.2: An applet at work in the AppletViewer

Handling Non-Java Browsers

"Uh-oh," says the Novice Programmer, "there is a problem. Some users are using a slightly nonstandard web browser." "That does sound slightly nonstandard," you say. "And," the NP continues, "it doesn't support Java. Is there any way of informing the users that they are missing something?" "There sure is," you say. "Pull up a chair and we'll go through it."

If you enclose text inside an `<APPLET>` tag, that text will be displayed if the web browser does not support Java. For example, here's how you might alert users that they are missing your applet:

```
<APPLET CODE = "CreateApplet.class" WIDTH = 100 HEIGHT = 100>
  Sorry, you don't have Java, so you can't see my beautiful applet.
</APPLET>
```

Embedding **<APPLET>** Tags in Code

The Novice Programmer sighs. "I have been writing applets, and it's always a little bit of a pain." "What is?" you ask. "Writing a web page with an `<APPLET>` tag to test the applet out—isn't there an easier way?" "Sure," you say, "you can embed the `<APPLET>` tag directly in the source code file."

The developers at Sun realized that sometimes it's annoying to create a web page to test an applet; so if you use the Sun appletviewer, you can now place an `<APPLET>` tag directly into the `EmbedAppletTag.java` file in a comment, like this:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
CODE=EmbedAppletTag.class
WIDTH=300
HEIGHT=200 >
</APPLET> */
public class EmbedAppletTag extends Applet {
        public void paint(Graphics g) {
  g.drawString("Kogent Learning Solutions!", 80, 100); }
}
```

After creating the `.class` file, you can now start the appletviewer with the `.java` file directly:

```
C:\>appletviewer EmbedAppletTag.java
```

Using the init, start, stop, destroy, paint, and update Methods

The Novice Programmer is back and says, "My browser has gone all wacky—it draws my applets in gray!" "That's the default for many web browsers," you say. "However, you can change that by adding some initialization code to your applet in the `init()` method."

Here are a number of important applet methods you should know about:

- `init()`—This is the first method to be called, and it is called only once. You initialize the applet here.
- `start()`—This method is called after `init`, and it is called each time an applet appears again on the screen. That is, if the user moves to another page and then comes back, the `start` method is called again.
- `stop()`—This method is called when the browser moves to another page. You can use this method to stop additional execution threads your applet may have started.
- `destroy()`—This method is called when the applet is about to be removed from memory. You can perform cleanup here.
- `paint()`—This method is called when the applet is to be redrawn. It is passed an object of the `Graphics` class, and you can use that object's methods to draw in the applet.
- `update()`—This method is called when a portion of the applet is to be redrawn. The default version fills the applet with the background color before redrawing the applet, which can lead to flickering when you are performing animation, in which case you would override this method.

You can override these methods to customize them as you like. We have already overridden the `paint` method to draw a string of text; here, we override the `init()` method to change the background color of the applet to orange, using the applet `setBackground()` method and passing it the `orange` field of the Java `Color` class. This applet also implements other methods listed previously. Here's the code:

```
import java.applet.Applet;
import java.awt.*;
/* <APPLET
CODE=AppletMethods.class
WIDTH=300
HEIGHT=200 >
</APPLET> */
public class AppletMethods extends Applet {
        public void init() { setBackground(Color.orange); }
        public void start() { }
        public void paint(Graphics g) {
        g.drawString("Kogent Learning Solutions!", 80, 100); }
        public void stop() { }
        public void destroy() { }
}
```

The `init()` method is a very useful one, and it's commonly overridden because it lets you initialize your applet. In this case, we have changed the background color from gray (the default in many browsers) to orange. Applets excel at graphics; the next topic provides an overview of handling graphics in applets.

Drawing Graphics in Applets

You'll learn much more about drawing in applets in a few chapters, but it's worth covering some of the graphics methods you'll use here:

- `paint()`—It is called when the applet is to be redrawn.
- `Repaint()`—It calls this method to force the applet to be painted.
- `drawString()`—It draws a string of text.
- `setBackground()`—It sets the background color.
- `setForeground()`—It sets the foreground (drawing) color.
- `draw3DRect()`—It draws a 3D rectangle.
- `drawBytes()`—It draws text, given a byte array.
- `drawImage()`—It draws an image.
- `drawOval()`—It draws an oval (including circles).
- `drawPolyLine()`—It draws a line with multiple segments.
- `drawRoundRect()`—It draws a rounded rectangle.
- `drawArc()`—It draws an arc.
- `drawChars()`—It draws text, given a character array.
- `drawLine()`—It draws a line.
- `drawPolygon()`—It draws a polygon.
- `drawRect()`—It draws a rectangle.

Two methods that are particularly worth noticing are the `paint()` method, in which you paint the applet, and the `repaint()` method, which forces the `paint()` method to be called again.

Reading Parameters in Applets

The Big Boss appears, chomping on a cigar, and says, "We need to personalize the greeting in our applet by customer." "But there are thousands of customers," you say. "We can't recompile the applet for each one and store each new version on the website." "What do you suggest?" the BB asks.

What will you say?

You can pass parameters to applets in the `<APPLET>` tag, and the applet code can read the values of those parameters, which means that to customize the applet, you only need to supply different parameters in the `<APPLET>` tag. To actually get the value of a parameter, you use the `Applet` class's `getParameter()` method, passing it the name of the parameter as specified in the `<PARAM>` tag. The `getParameter()` method will return the value of the parameter that was set in the `<PARAM>` tag. Here's an example in which we pass a parameter named `string` to an applet; the value of this parameter is the text that the applet should display. Here's how this looks in code:

```
import java.applet.Applet;
import java.awt.*;
/* <APPLET
   CODE=AppletParameters.class
   WIDTH=300
   HEIGHT=200 >
   <PARAM NAME = string VALUE = "Kogent Learning Solutions!">
</APPLET> */
public class AppletParameters extends Applet {
        public void paint(Graphics g) {
   g.drawString(getParameter("string"), 80, 100); }
}
```

You can find the process of Command-Line Arguments Passed to main method in detail in Chapter 4.

Using Java Consoles in Browsers

"All this `drawString()` stuff is OK," the Novice Programmer says, "but what about the Java console? What if I use `System.out.println()` in an applet?" "That depends," you say, "on your browser."

Here's an applet that displays a message and uses `System.out.println()` to print to the console:

```
import java.applet.Applet;
import java.awt.*;
/* <APPLET
   CODE=AppletAndConsole.class
   WIDTH=300
   HEIGHT=200 >
</APPLET> */
public class AppletAndConsole extends Applet {
         public void paint(Graphics g) {
   g.drawString("Kogent Learning Solutions!", 80, 100);
   System.out.println("Kogent Learning Solutions!"); }
}
```

If you run this applet with the Sun appletviewer, the applet will open in a separate window, and you'll see "Kogent Learning Solutions!" in the console window. Web browsers often have a Java Console as well, although you often have to enable them before using. The way to enable the Java Console differs, unfortunately, from browser to browser as well as from version to version. Currently, you enable the Java Console in Internet Explorer by selecting Tools | Internet Options, clicking the Advanced tab, and selecting the Java Console Enabled checkbox. And you can open the Java Console by selecting the View | Java Console

Figure 6.3 shows the result of the preceding applet as it appears in the Internet Explorer Java Console, which pops up when you print to the applet. In Netscape Navigator, you can open the Java Console with the Communicator | Java Console menu item.

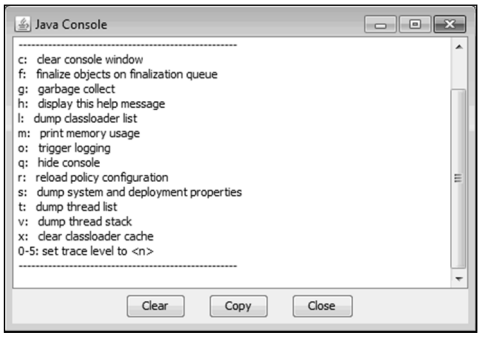


Figure 6.3: Using Internet Explorer's Java console

Adding Controls to Applets: Text Fields

"OK," says the Novice Programmer. "I can draw text in an applet now. But what if I want to let the user *enter* some text?" "For that," you say, "you can use all kinds of text controls, such as text fields."

Text fields are about the most basic controls you can use in AWT, so they provide a good starting point. A text field displays text on a single line and lets the user edit that text.

We'll discuss text fields formally in the next chapter, but we'll put them to use here, too, when talking about event handling. Here's a text field example in which we are creating a text field 20 characters wide in an applet's `init()` method (note that we are importing AWT classes to be able to use text fields):

```
import java.applet.Applet;
import java.awt.*;
/* <APPLET
   CODE=AppletControls1.class
   WIDTH=300
   HEIGHT=200 >
</APPLET> */
public class AppletControls1 extends Applet {
         public TextField text1;
         public void init() {
                  text1 = new TextField(20);
```

```
    . . . }
}
```

After creating a new control, you must add it to the applet so that it's displayed. Here's an example:

```
public void init() {
   text1 = new TextField(20); add(text1);
   . . .
}
```

The `add` method adds the control to the current layout manager, which decides where the control should be placed (you'll see the details on layout managers in the next chapter). Now that the text field has been added to the applet, we can place the text "Kogent Learning Solutions!" in the text field with the `setText()` method, like this:

```
public void init() {
text1 = new TextField(20);
add(text1);
text1.setText("Kogent Learning Solutions!");
}
```

The result of this code appears in Figure 6.4, where you can see the text field with the message we have put into it. The user can also edit this text. We'll take a closer look at text fields in the next chapter. Another basic control is the AWT Button control; we'll use buttons and text fields to discuss event handling, so we'll introduce the Button control in the next topic. You'll learn about buttons and text fields in detail in the next chapter.

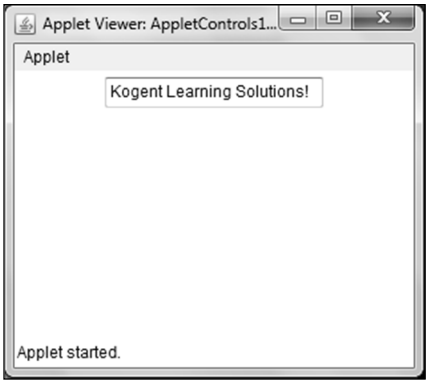


Figure 6.4: Adding a Text Field to an applet

Adding Controls to Applets: Buttons

"I'm ready for the next step," the Novice Programmer reports. "I have added text fields to my applets, what's next?" "Buttons," you say. "Pull up a chair and let's talk about them."

Users can click buttons in their applet to signal that they want to perform some action; for example, you may have a button labeled "Change Color" that, when clicked, changes the background color of the applet using the `setBackground()` method. Buttons are supported in the `java.awt.Button` class, which we'll discuss in detail in the next chapter. It's easy enough to add a button to an applet— you do it in much the same way as adding a text field to an applet, as demonstrated in the previous example. Here, we are creating and adding a button with the caption "Display Text".

```
public class AppletControls2 extends Applet {
TextField text1; Button button1;
public void init() {
text1 = new TextField(20); add(text1);
button1 = new Button("Display Text"); add(button1);
   }
```

The result of this code appears in Figure 6.5, where you can see the text field and a button with caption "Display Text." The real trick is to get something to happen when the user clicks the button, and for that, we'll have to take a look at event handling (see the next topic).

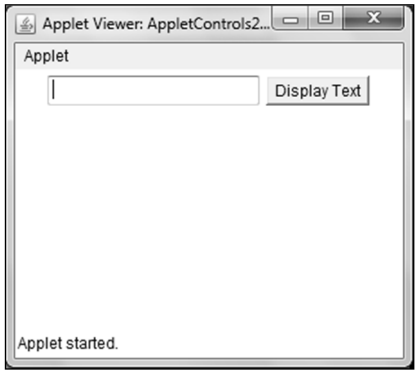


Figure 6.5: Adding a Button to an applet

Handling Events

"Hey," says the NP, "I have put a Button in my applet; but when I click it, nothing happens. What gives?" "What gives," you say, "is that you have to implement event handling."

Event handling—the process of responding to button clicks, mouse movements, and so on—has become a complex topic in Java. Starting with Java 1.1, event handling changed significantly in Java; the current model is called *delegated event handling*. In this event handling model, you must specifically *register* with Java if you want to handle a specific event such as a button click (the idea is that performance is improved if only code that needs to handle specific events is informed of those events, and not the rest of your code).You register for events by implementing an event listener interface. Here are the available event listeners and the kinds of events they handle:

- `ActionListener`—It listens for action events, such as button clicks
- `AdjustmentListener`—It listens for adjustment events, such as scrollbar movements
- `ComponentListener`—It listens for cases in which a component is hidden, moved, resized, or shown
- `ContainerListener`—It listens for cases in which a component is added or removed from a container
- `FocusListener`—It listens for cases in which a component gains or loses the focus
- `ItemListener`—It listens for cases in which the state of an item changes
- `KeyListener`—It listens for keyboard events
- `MouseListener`—It listens for cases in which the mouse is clicked, enters a component, exits a component, or is pressed
- `MouseMotionListener`—It listens for cases in which the mouse is dragged or moved
- `MouseWheelListener`—It listens for cases in which the mouse wheel event is rotated in a component
- `TextListener`—It listens for text value changes
- `WindowListener`—It listens for cases in which a window is activated, deactivated, iconified, deiconified, opened, closed, or quit

Each listener is an interface, and it's up to you to implement the methods of the interface (for more on interfaces, see the previous chapter). Each of these methods is passed a type of event object that corresponds to the kind of event:

- `ActionEvent`—It handles buttons, list double-clicks, and menu item clicks
- `AdjustmentEvent`—It handles scrollbar movements
- `ComponentEvent`—It handles cases in which a component is hidden, moved, resized, or becomes visible
- `ContainerEvent`—It handles cases in which a component is added or removed from a container
- `FocusEvent`—It handles cases in which a component gains or loses the focus

- `InputEvent`—It handles checkbox and list item clicks, choice control selections, and checkable menu item selections
- `ItemEvent`—It handles cases when an item is selected, de-selected, expanded, or contracted
- `KeyEvent`—It handles input from the keyboard
- `MouseEvent`—It handles cases in which the mouse is dragged, moved, clicked, pressed, released, or entered or exits a component
- `MouseWheelEvent`—It handles the cases when the mouse wheel is rotated in a component
- `TextEvent`—It handles the value of a text field or text area being changed
- `WindowEvent`—It handles cases in which a window is activated, deactivated, iconified, deiconified, opened, closed, or quit

Standard Event Handling

It's time to put some of what we have covered to work. We'll start by adding a new button with the text "Display Text" to an applet as well as adding an action listener that will be notified when the button is clicked. To add an action listener to the button, you use the button's `addActionListener()` method, passing it an object that implements the methods of the `ActionListener` interface. This object can be an object of the applet's `main` class or another class. We'll go over both variations here, starting with sending event notifications to the `main` applet class.

TIP

You can also use removeListener() method to unregister an event listener.

Here's how we add an action listener to a button, sending event notifications to the current applet object (note that we indicate that the `AppletEvent1` class now implements the `ActionListener` interface):

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
CODE=AppletEvent1.class
WIDTH=300
HEIGHT=200>
</APPLET> */
public class AppletEvent1 extends Applet implements ActionListener {
TextField text1;
Button button1;
public void init() {
text1 = new TextField(20);
add(text1);
button1 = new Button("Display Text");
add(button1);
button1.addActionListener(this);
.... }
}
```

Now we have to implement the methods of the `ActionListener` interface. It turns out that this interface has only one method `actionPerformed()`, which is passed an object of the `ActionEvent` class when the button is clicked:

```
void actionPerformed(ActionEvent e)
```

`ActionEvent` objects (which we'll discuss in the next chapter) inherit a method named `getSource()` from the `EventObject` class, and this method returns the object that caused the event. That means we can check whether this event was caused by the button, `button1`, and if so, place the text "Kogent Learning Solutions!" into the text field `text1`, like this:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
```

```
CODE=AppletEvent1.class
WIDTH=300
HEIGHT=200 >
</APPLET> */
public class AppletEvent1 extends Applet implements ActionListener {
TextField text1; Button button1;
public void init() {
   text1 = new TextField(20);
   add(text1);
   button1 = new Button("Display Text");
   add(button1);
   button1.addActionListener(this); }
   public void actionPerformed(ActionEvent event) {
   String text = new String ("Kogent Learning Solutions!");
   if(event.getSource() == button1) { text1.setText(text); }
}
}
```

This applet appears in Figure 6.6. When you click the button, the text "Kogent Learning Solutions!" appears in the text field.

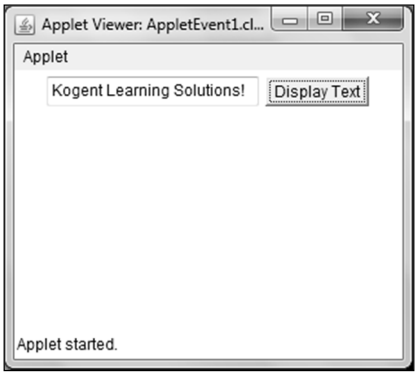


Figure 6.6: Supporting button clicks

Now we can implement the methods of the `FocusListener` interface. It turns out that this interface has methods, like `focusGained()` and `focusLost()`, which is passed an object of the `FocusEvent` class when the tab is pressed:

```
void focusGained(FocusEvent e)
```

To control the focus, you can use `setFocusable()`, which requires Boolean value, i.e., true or false and will decide whether the component can gain focus or not, and also use `requestFocusInWindow()`, which will let the component to gain the focus on startup.

That means we can check whether this event was caused by the buttons, `button1`, `button2`, or `button3`, and if so, place the text "Focus is on Button2!" into the text field `text1` when focus is on `button2`, like this:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
   CODE=AppletEvent2.class
   WIDTH=200
   HEIGHT=200 >
</APPLET> */
public class AppletEvent2 extends Applet implements FocusListener, MouseWheelListener
{
         TextField text1; Button button1, button2, button3;
         public void init() {
text1 = new TextField(20);
add(text1);
button1 = new Button("Button1");
button2 = new Button("Button2");
```

```
    button3 = new Button("Button3");
    add(button1);
    add(button2);
    add(button3);
    button1.addFocusListener(this);
    button2.addFocusListener(this);
    button3.addFocusListener(this);
    button1.setFocusable(false);
    button2.requestFocusInWindow();
    text1.addMouseWheelListener(this);
    }
            public void focusGained(FocusEvent event) {
    String text = new String ("Focus is on Button2!");
    if(event.getSource() == button2) {
    text1.setText(text); transferFocusUpCycle(); }
    if(event.getSource() == button3) { transferFocusBackward(); }
            }
            public void focusLost(FocusEvent e) { }
            public void mouseWheelMoved(MouseWheelEvent e) { }
    }
```

This applet appears in Figure 6.7. When you start the applet, the text "Focus is on Button2!" appears in the text field. We can change the focus by using tab, but focus will not go to `button1` because of `isFocuable(false)` for `button1`.



Figure 6.7: Adding Listeners and Buttons to the Applet

The class you register to listen for events doesn't need to be the main applet class (and in fact, the Sun developers originally intended that it not be, although it's now become common practice). We'll take a look at using other classes to listen for events next.

Using Delegated Classes

Here's an example in which we are creating a new class to implement the `ActionListener` interface. Note that this is a little clumsy. We want to work with the text field and button controls in the main applet object, so we have to pass and store references to that object in the new class's constructor:

```
    import java.applet.Applet;
    import java.awt.*;
    import java.awt.event.*;
    /* <APPLET
    CODE=AppletEvent3.class
    WIDTH=200
    HEIGHT=200 >
    </APPLET> */
    public class AppletEvent3 extends Applet {
    public TextField text1;
    public Button button1;
    public void init() {
```

```
            text1 = new TextField(20);
            add(text1);
      button1 = new Button("Display Text");
      add(button1);
      MyListener ml = new MyListener(this);
      button1.addActionListener(ml); }
   }
   class MyListener implements ActionListener {
      AppletEvent3 e3;
      MyListener(AppletEvent3 ae3) { e3 = ae3; }
      public void actionPerformed(ActionEvent event) {
   String text = new String ("Kogent Learning Solutions!");
   if(event.getSource() == e3.button1) { e3.text1.setText(text); }
   }
   }
```

This code works the same as the previous version of this applet, except that internally it uses a new class to handle events, not the main applet class. There are times when this is useful, such as when you have many events to handle and don't want to make your main applet class huge.

Here's another example in which we are creating a new class to implement the `FocusListener` interface. In this example, we are using a concept of changing the usual behavior of the key. To see the actual working of this code, use the Tab key to navigate through the buttons and text field instead of the mouse.

Note that this is a little clumsy, as a lot of packages are imported to work with keyboard events. We want to work with the text field and button controls in the main applet object, so we have to pass and store references to that object in the new class's constructor:

```
   import java.applet.Applet;
   import java.awt.*;
   import java.awt.event.*;
   import java.awt.Container;
   import java.awt.FocusTraversalPolicy;
   import java.awt.KeyboardFocusManager;
   import java.awt.event.KeyEvent;
   import java.util.HashSet;
   import java.util.Set;
   import javax.swing.KeyStroke;
   /* <APPLET
      CODE=AppletEvent4.class
      WIDTH=200
      HEIGHT=200 >
   </APPLET> */
   public class AppletEvent4 extends Applet implements FocusListener {
            public TextField text1; public Button button1,button2;
            public void init() {
            this.setFocusTraversalKeysEnabled(true);
            text1 = new TextField(20);
            add(text1);
            button1 = new Button("Button1");
            button2 = new Button("Button2");
            add(button1);
            add(button2);
            button1.addFocusListener(this);
            button2.addFocusListener(this); }
            public void focusGained(FocusEvent event) {
      if(event.getSource() == button1) { text1.setText("Focus on Button1"); }
      if(event.getSource() == button2) { text1.setText("Focus on Button2"); }
            }
            public void focusLost(FocusEvent e) { }
      }
```

This applet appears in Figure 6.8. We can use Tab key to shift the focus. In addition, there are other ways of determining which object caused the event; for example, you can use *commands*.



Figure 6.8: Key Traversal and Focus Listener

Using Action Commands

Java lets you associate commands with events caused by AWT buttons and menu items. When working with buttons, the command is just the caption of the button by default (you'll see how to create custom commands in the next chapter), so that you can determine which button was clicked by looking at its caption (not a good idea if your program changes captions). You can get the command for a button with the `getActionCommand()` method. Here's how we implement the preceding applet using commands:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
CODE=AppletEvent5.class
WIDTH=200
HEIGHT=200 >
</APPLET> */
public class AppletEvent5 extends Applet implements ActionListener {
TextField text1; Button button1;
public void init() {
   text1 = new TextField(20);
   add(text1);
   button1 = new Button("Display Text");
   add(button1);
   button1.addActionListener(this); }
   public void actionPerformed(ActionEvent event) {
   String text = new String ("Kogent Learning Solutions!");
   String caption = event.getActionCommand();
   if(caption.equals("Display Text")) { text1.setText(text); }
}
}
```

This introduces you to the process of handling events the modern way. However, the old Java 1.0 way still works in Java, although it's considered deprecated. For the sake of completeness, we'll take a look at it in the next topic.

Handling Events the Old Way

Java 1.0 uses a non-delegated approach to events. In the Java 1.0 event model, you don't need to register to get events—they are just passed to you anyway, and you can handle them in a method named `action`. For example, here's how the previous applet looks using the old event model:

```
import java.applet.Applet;
import java.awt.*;
/* <APPLET
CODE=AppletEvent6.class
WIDTH=200
HEIGHT=200 >
</APPLET> */
public class AppletEvent6 extends Applet {
```

```
    TextField text1; Button button1;
    public void init() {
  text1 = new TextField(20);
  add(text1);
  button1 = new Button("Display Text");
  add(button1); }
  public boolean action (Event e, Object o) {
  String caption = (String)o; String text = "Kogent Learning Solutions!";
  if(e.target instanceof Button) {
  if(caption == "Display Text") { text1.setText(text); }
  }
  return true; }
  }
```

The problem was that the `action()` method often became huge, so the Java designers introduced the delegated event model, in which you can send events where you want and are not restricted to an `action()` method. In fact, there's one more way of handling events—by extending components.

Extending Components

If you like being sneaky, you can handle events by deriving new classes from components and overriding the methods in the components that handle events. In fact, this way of doing things ends up much like the Java 1.0 technique of event handling because all event handling takes place in one method; therefore, it's discouraged. However, we'll cover it here to be complete.

Here's how we implement the applet by extending the `Button` class in a derived class named `NewButton`. In this new class, we override the `Button` class's `processActionEvent()` method, displaying the text "Kogent Learning Solutions!" in the applet's text field after calling the `Button` class's `processActionEvent()` method:

```
  import java.applet.Applet;
  import java.awt.*;
  import java.awt.event.*;
  /* <APPLET
  CODE=AppletEvent7.class
  WIDTH=200
  HEIGHT=200 >
  </APPLET> */
  class NewButton extends Button {
  AppletEvent7 ae;
  NewButton(AppletEvent7 ref, String s) {
  super(s); ae = ref;
  enableEvents(AWTEvent.ACTION_EVENT_MASK); }
  protected void processActionEvent (ActionEvent e) {
  super.processActionEvent(e);
  ae.text1.setText("Kogent Learning Solutions!"); }
  }
```

Now all we have to do is to create a new button of this class and add it to the applet:

```
  public class AppletEvent7 extends Applet {
  TextField text1;
  Button button1;
  public void init() {
  text1 = new TextField(20);
  add(text1);
  button1 = new NewButton(this, "Display Text");
  add(button1); }
  }
```

That's all it takes; now this applet works like the others.

Using Adapter Classes

"Ugh!" says the Novice Programmer. "I understand about the delegated event model that uses listener interfaces, but sometimes it's a pain to have to implement all the methods of an interface when you just want to use one method." "That's true," you say, "which is why Sun introduced adapter classes to make the whole process much easier."

Adapter classes are classes that have already implemented the various event interfaces available. Each method of an interface is implemented as an empty method without any code in an adapter class, and all you need to do is override the method or methods you want to provide code for.

Let's look at an example. In this case, we'll start with an applet that stores a string to print, which is "Kogent Learning Solutions!" by default, and displays that string in the `paint()` method. Then we are going to add a mouse listener (which you'll learn more about in the next chapter) to the applet; so when the user clicks the applet, the message "Hello from Java!" appears. Unlike the `ActionListener` interface, the `MouseListener` interface has five methods that must be implemented. However, we just want to use the `mouseClicked()` method to handle mouse clicks, so we'll use the `MouseAdapter` class instead.

We start by adding a class that implements a mouse listener to the program. This class will subclass the `MouseAdapter` class, and we'll call it `MyAdapter`:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
        CODE=AppletAdapter.class
        WIDTH=300
        HEIGHT=200 >
</APPLET> */
public class AppletAdapter extends Applet {
public String text = "Kogent Learning Solutions!";
public void init() { addMouseListener(new MyAdapter(this)); }
public void paint(Graphics g) { g.drawString(text, 80, 100); }
}
```

We pass an object of the main applet class to the `MyAdapter` class's constructor, so it can reach the fields in the applet. When the mouse is clicked, the text string in the applet is replaced with the text "Hello from Java!" and then the applet is repainted, causing the new string to appear on the screen:

```
class MyAdapter extends MouseAdapter {
AppletAdapter aa;
MyAdapter(AppletAdapter appletobject) { aa = appletobject; }
public void mouseClicked(MouseEvent e) {
aa.text = "Hello from Java!";
aa.repaint(); }
public void mouseReleased(MouseEvent e) { }

}
```

The result of this code appears in Figure 6.9 and Figure 6.10.

Note that this was a little awkward because we had to pass the applet object to the `MyAdapter` class's constructor so that the class could reach the fields in the `AppletAdapter` class. On the other hand, inner classes have access to the fields of their enclosing classes, so adapters are often implemented as inner classes. We'll take a look at that next.

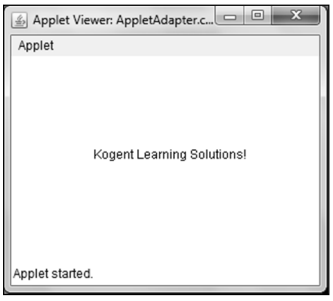


Figure 6.9: Supporting button clicks with adapter classes



Figure 6.10: Supporting button release with adapter classes

Using Anonymous Inner Adapter Classes

You first saw anonymous inner classes in the previous chapter. The format for these classes is:

```
new SuperType(constructor parameters) { } //methods and data
```

where *SuperType* is the class or interface from which the anonymous inner class is subclassed. In this case, we are going to derive an anonymous inner class from the `MouseAdapter` class to implement the applet you first saw in the previous topic. Here's how that looks (note that we are overriding the `mouseClicked()` method as we did in the previous topic, but this time we are using an anonymous inner class, which makes the code much more compact):

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
        CODE=AppletAnonyAdapter.class
        WIDTH=300
        HEIGHT=200 >
</APPLET> */
public class AppletAnonyAdapter extends Applet {
public String text = "Kogent Learning Solutions!";
public void init() {
addMouseListener(new MouseAdapter() {
                public void mousePressed(MouseEvent me) {
  text = "Hello from Java!"; repaint(); }});
        }
        public void paint(Graphics g) { g.drawString(text, 80, 100); }
}
```

Creating Windowed Applications

The NP appears and says. "OK, I can create applets now, but what about applications that use windows?" "Coming right up," you say.

Unlike writing applets, when you write a windowed application, you are responsible for creating your own window in which to display that application. The most common type of window to use for this purpose is the Java frame window, as supported in the `java.awt.Frame` class. Here's the inheritance diagram for that class (this class is derived from the `java.awt.Window` class, which we'll cover in a few chapters):

```
java.lang.Object
   |____java.awt.Component
         |____java.awt.Container
                |____java.awt.Window
                       |____java.awt.Frame
```

You'll find the constructors for the `Frame` class in Table 6.4 and its methods in Table 6.5.



Although the Applet class's default layout manager is the flow layout manager, the default layout manager in the Frame class is the border layout manager. If you want to use another layout when adding components to a Frame window, you'll have to change the layout manager yourself.

Table 6.4: Constructors of the Frame class

Constructor	Does this
`Frame()`	It constructs a new instance of `Frame` that's initially invisible
`Frame(GraphicsConfiguration gc)`	It constructs a new `Frame` with specified `GraphicsConfiguration`
`Frame(String title)`	It constructs a new `Frame` object that's invisible with the given title
`Frame(String title, GraphicsConfiguration gc)`	It constructs a new `Frame` with specified title and `GraphicsConfiguration`

Table 6.5: Methods of the Frame class

Method	Does this
`void addNotify()`	It makes this frame displayable by connecting it to a native screen resource
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this frame
`int getCursorType()`	It is deprecated and replaced by `Component.getCursor()`
`int getExtendedState()`	It gets the state of the frame
`static Frame[] getFrames()`	It gets an array containing all frames created by the application
`Image getIconImage()`	It gets the image to be displayed as icon for the frame
`Rectangle getMaximizedBounds()`	It gets the maximized bounds for the frame
`MenuBar getMenuBar()`	It gets the menu bar
`int getState()`	It gets the state of the frame
`String getTitle()`	It gets the title of the frame
`boolean isResizable()`	It indicates whether this frame is resizable by the user
`boolean isUndecorated()`	It indicates whether this frame is undecorated
`protected String paramString()`	It returns a string that represents the state of the frame
`void remove (MenuComponent m)`	It removes the specified menu bar from the frame
`void removeNotify()`	It makes this frame undisplayable by removing its connection to its native screen resource
`void setBackground (Color bgColor)`	It sets the indicated background color of the frame
`void setCursor (int cursorType)`	It is deprecated and replaced by `Component.setCursor(Cursor)`
`void setExtendedState(int state)`	It sets the state of the frame
`void setIconImage (Image image)`	It sets the image to be displayed as icon for the frame
`void setMaximizedBounds(Rectangle bounds)`	It sets the maximized bounds for the frame
`void setMenuBar(MenuBar mb)`	It sets the menu bar for the frame to the specified menu bar
`void setOpacity(float opacity)`	It sets the opacity of the window
`void setResizable (boolean resizable)`	It sets whether the frame is resizable by the user
`void setShape(Shape shape)`	It sets the shape of the window
`void setState(int state)`	It sets the state of the frame
`void setTitle(String title)`	It sets the title for the frame to the specified string
`void setUndecorated(boolean undecorated)`	It enables or disables the frame decorations

Here's an example in which we derive a new class, `AppFrame`, from the `Frame` class and customize this new class to display the string "Kogent Learning Solutions!" to make it look like the applet developed earlier.

As with the `java.applet.Applet` class, the `java.awt.Frame` class is derived from the `java.awt.Window` class, so we can use the `paint()` method to display graphics in the `Frame` class. In fact, the `paint()` method will look just as it did in the applet created earlier in the application's code file `Application1.java`. Here's the code:

```
import java.awt.*;
class AppletFrame extends Frame {
   public void paint(Graphics g) {
   g.drawString("Kogent Learning Solutions!", 80, 100); }
}
```

A `main()` method is needed to start the application itself, so we create that method in a new class named `Application1`:

```
public class Application1 {
public static void main(String [] args) { . . . }
}
```

To display the application's frame window, we create an object of the `AppletFrame` class, like this:

```
public class Application1 { public static void main(String [] args) {
AppletFrame f = new AppletFrame();
. . . }
}
```

Now we give the frame window object a size of 300 by 200 pixels and display it on the screen with the `setVisible(true)` method:

```
public class Application1 {
public static void main(String [] args) {
AppletFrame f = new AppletFrame();
f.setSize(300,200);
f.setVisible(true); }
}
```

TIP

If you don't give a frame window a size before displaying it, all you'll see is a title bar. In Graphical User Interface (GUI) terms, there will be no client area, which is the area under the title bar in which the application displays itself.

The result of this code appears in Figure 6.11. As you can see, the message "Kogent Learning Solutions!" appears in the application, as intended. You can also add controls to this application, just as we did with the button applet earlier in this chapter.

There are a few things to mention here: First, if you launch the application with the `java` tool (that is, as `java Application`), that tool will wait until the application ends before returning control to the console. From the user's point of view, the console appears to hang until the application is exited. If you want control to return immediately to the console after the application is launched and displays its own window, use the `javaw` tool instead.

Another important point is that if you want to distribute your applications to users who don't have the SDK installed, you can use the Java Runtime Environment (JRE).

Finally, it's important to point out that there's no easy way to exit the application in Figure 6.11—clicking the close button has no effect at all. You can press Ctrl+C to end execution of the `java` tool in the console window, but that's pretty awkward. Instead, you have to handle window-closing events to end the application when its window is closed. We'll take a look at this in the next topic related to exiting an application when window is closed.



Figure 6.11: Creating a windowed application

Before going to next topic, we'll take some more examples on creating windowed application to show the frame creation with `createVolatileImage()` method. Here is one:

```
import java.awt.Color;
import java.awt.Component;
import java.awt.Frame;
```

```
import java.awt.Graphics;
import java.awt.image.VolatileImage;
public class Application2 extends Component {
  public VolatileImage buffer = null;
  void createBackBuffer() {
        if (buffer != null) {
                buffer.flush();
                buffer = null; }
        buffer = createVolatileImage(getWidth(), getHeight()); }
        public void paint(Graphics g) {
        if (buffer == null) { createBackBuffer(); }
        do {
                int code = buffer.validate(getGraphicsConfiguration());
                if (code == VolatileImage.IMAGE_RESTORED) {}
                else if (code == VolatileImage.IMAGE_INCOMPATIBLE) {
                createBackBuffer(); }
        Graphics graph = buffer.getGraphics(); graph.setColor(Color.white);
        graph.fillRect(0, 0, getWidth(), getHeight());
        graph.setColor(Color.red);
        graph.drawLine(0, 0, getWidth(), getHeight());
        g.drawImage(buffer, 0, 0, this); }
while (buffer.contentsLost());
}
  public static void main(String args[]) {
        Frame frame = new Frame();
        frame.setSize(300, 200);
        frame.add(new Application2());
        frame.setVisible(true); }
}
```

The result of the code appears in Figure 6.12.

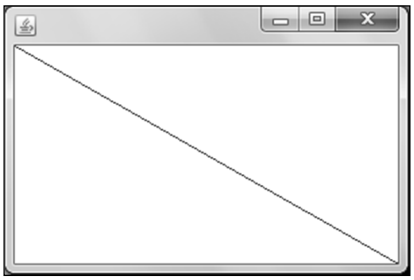


Figure 6.12: Creating windowed application with frame creation

Here is another example of frame creation using `getPreferredSize()` method:

```
import java.awt.*;
import java.awt.event.*;
public class Application3 extends Frame implements ItemListener {
  private Choice choice; Label label = new Label("Make a choice");
  Application3() {
  super("Choices");
  add(label, BorderLayout.SOUTH);
  choice = new Choice();
  choice.addItem("One");
  choice.addItem("Two");
  choice.addItem("Three");
  choice.addItemListener(this);
  add(choice, BorderLayout.CENTER); pack();
  setSize(200,getPreferredSize().height);
  setLocation(25,25); setVisible(true); }
  public void itemStateChanged(ItemEvent ie) {
                String state = "Deselected";
```

```
      if(ie.getStateChange() == ItemEvent.SELECTED)
      state = "Selected";
      label.setText(ie.getItem() + " " + state ); }
      public static void main(String args[]) {
      new Application3(); }
   }
```

The result of the code appears in Figure 6.13.

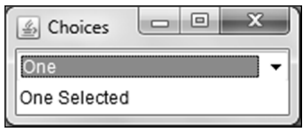


Figure 6.13: Creating windowed application using setPreferredSize()

Exiting an Application When Its Window Is Closed

"Hey," says the NP, "I found it a little difficult to end my first windowed application—clicking the close button didn't do anything." "So how did you end the application?" you ask. "Turned off my computer," the NP says sheepishly.

Java expects you to handle the case in which the user clicks the close button in an application's window yourself (although this has been made easier with Swing windows). To end an application when the user clicks the close button, you must catch window events, which you can do with the `WindowListener` interface. We'll show a compact way of doing that here by modifying the application developed in the previous chapter using the `WindowAdapter` class, which implements the `WindowListener` interface with an empty implementation of each method in `WindowListener`. In this case, we'll use an anonymous inner adapter class (see the topic "Using Anonymous Inner Adapter Classes," earlier in this chapter, for more details) and override the `windowClosing()` method. In that event's method, we'll add code to exit the application—`System.exit(0)`. This ends the application with an exit code of 0 (which means a normal termination). Here's the code:

```
   import java.awt.*;
   import java.awt.event.*;
   class AppletFrame extends Frame {
           public void paint(Graphics g) {
     g.drawString("Kogent Learning Solutions!", 80, 100); }
   }
   public class ExitApplication {
     public static void main(String [] args) {
   AppletFrame f = new AppletFrame();
   f.setSize(300, 200);
   f.addWindowListener(new WindowAdapter() {
     public void windowClosing(WindowEvent e) { System.exit(0); }});
     f.setVisible(true); }
   }
```

That's all it takes. Now when you click the application window's close button, the window closes and the application exits.

Applications You Can Run as Applets

The Big Boss arrives and says, "We have got to cut development costs. From now on, all applets must also double as applications." "Hmm," you say, "when do you need that in place?" "Aren't you done yet?" the BB asks.

If you add a `main()` method to an applet, that applet can run as both an applet and an application; Java will ignore the `main()` method when it runs as an applet and will run the `main()` method when it runs as an application. Here's an example in which we have combined the applet and application into one program:

```
   import java.applet.Applet;
   import java.awt.*;
   import java.awt.event.*;
   /* <APPLET
```

```
            CODE=AppletApp.class
            WIDTH=200
            HEIGHT=200 >
    </APPLET> */
    public class AppletApp extends Applet {
            public static void main(String [] args) {
      AppletFrame f = new AppletFrame();
    f.setPreferredSize(new Dimension(WIDTH-0, HEIGHT-0));
    f.addWindowListener(new WindowAdapter() {public void
      windowClosing(WindowEvent e) {System.exit(0);}});
      f.setVisible(true);
      f.setIgnoreRepaint(false); }
            public void paint(Graphics g) {
                    g.drawString("Hello from Java!", 60, 100); }
    }
    class AppletFrame extends Frame {
            public void paint(Graphics g) {
      g.drawString("Kogent Learning Solutions!", 80, 100); }
    }
```

You can run this code as either an applet or application (the Big Boss would be proud).

Setting Applet Security Policies

"Hey," says the NP, "why can't I write a file from an applet?" "Because of security," you say, "but you can change that if you really want to." "I can!!," says the NP, "How??"

You can use the Java Policy Tool to change security settings. For example, you can allow applets to read and write files (normally they don't have access to files because when a user opens an applet in a browser, he or she might not expect you to start working with files on the system). Here's an example of an applet called `WriterApplet.java` that tries to write a file:

```
    import java.applet.Applet;
    import java.io.*;
    /* <APPLET
            CODE=WriterApplet.class
            WIDTH=300
            HEIGHT=200 >
    </APPLET> */
    public class WriterApplet extends Applet {
            public void init() {
      byte data[] = "Kogent Learning Solutions".getBytes();
      try {
      FileOutputStream fo = new FileOutputStream("file.txt");
      fo.write(data);
      fo.close(); }
    catch (Exception e) { System.out.println("Error: " + e); }
            }
    }
```

If you try to run this applet as it stands, you'll get a security violation error in the console window, and the applet won't start. However, you can change that with the Policy Tool, which lets you set security settings on an applet-by-applet and application-by-application basis.

NOTE

You must be very careful when adjusting Java's security settings—don't leave yourself open to security problems.

Start the Policy Tool now by typing `policytool` in the console window. The Policy Tool appears in Figure 6.14. To add a new policy entry for `WriterApplet.class`, click the Add Policy Entry button in the Policy Tool, which opens the Policy Entry dialog box, shown in Figure 6.15.



Figure 6.14: The Policy Tool

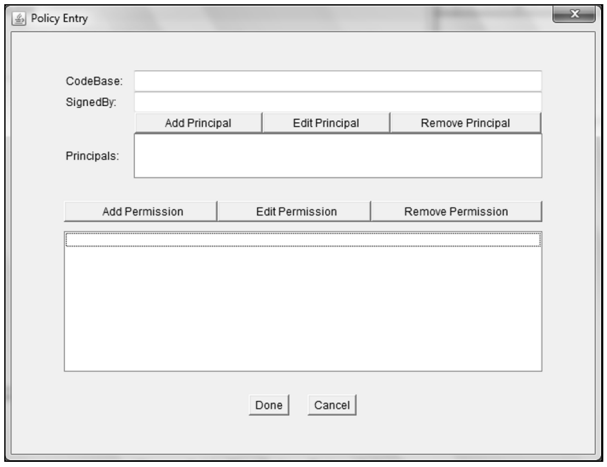


Figure 6.15: The Policy Entry dialog box

Enter the URL of the applet or application you want to set the security settings for (we'll use a `file:\\` URL here, but of course you can specify `HTTP://` URLs as well) and click the Add Permission button, which opens the Permissions dialog box, shown in Figure 6.16:

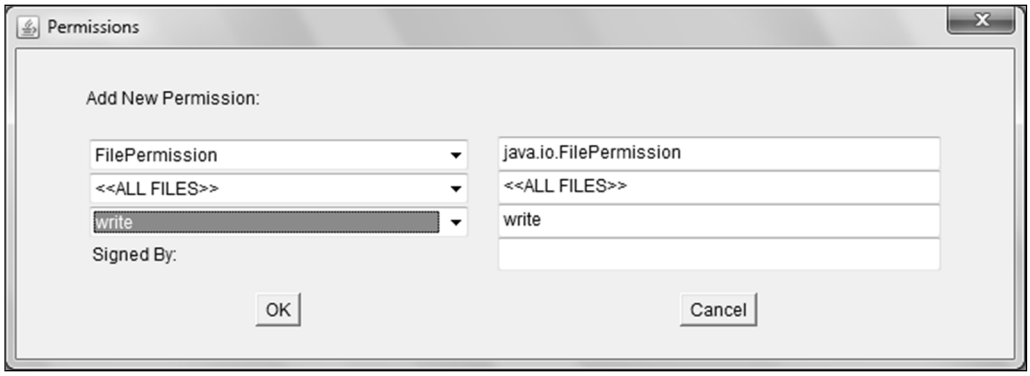


Figure 6.16: The Permissions dialog box

In this case, we want to set the file-writing permission for `WriterApplet.class`, so select `FilePermission`, `<<ALL FILES>>`, and write, as shown in Figure 6.16 (you don't have to fill in the Signed By line unless the code has been digitally signed). Now click OK to close the Permissions dialog box and then click "Done" to close the Policy Entry dialog box. You'll see the new policy entry in the Policy Tool, as shown in Figure 6.17.

In addition, you must save the new policy entry in a file, and you do that with the Save As menu item in the File menu. We'll save the policy entry in the file `D:\Chapter 6\java.policy` here, as you can see in Figure 6.17:

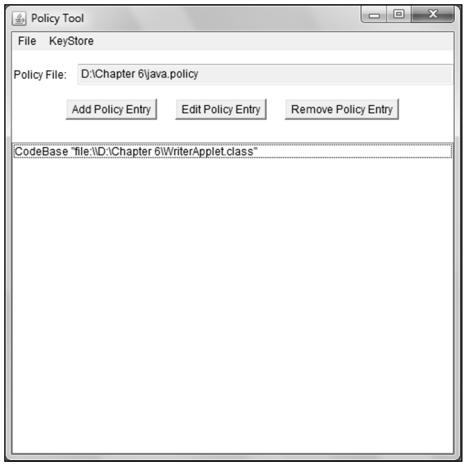


Figure 6.17: A new policy entry

That's all it takes. When you set the security policy for a program by providing that program's URL, you can grant all kinds of permissions to the program. In this case, the writer applet can now create `file.txt` and write this text to it.

Other Facilities in the **java.awt** Package

Java 6 added certain new features related to AWT. These features help in changing the look and feel of the Java Application GUI. These features are as follows:

- System Tray
- Splash Screen
- Dialog Modality
- Gif Writer
- Text Antialiasing

System Tray

It is a specific area that can be accessed during running programs. It appears at the bottom of the desktop. Any user can access it continually. All running application on the desktop can share this tray area. It is referred to as the Taskbar Status Area on the Microsoft Windows, the system tray on KDE, and the Notification Area on the Gnome Desktop. You can access the system tray in Java with two separate classes defined in `java.awt` package, i.e., `SystemTray` and `TrayIcon`.

These classes add some more functionality, such as graphics, pop-up menus, and floating tip functions to the system tray. The system tray contains method `getSystemTray()`. The system tray in Java can be accessed by calling the following method:

```
static SystemTray.getSystemTray()
```

Before accessing the system tray, the application always check the availability of system tray by using the `static SystemTray.isSupported()` method. This method returns false if it is found that the system tray is not present. In this case, if the `getSystemTray()` method is invoked by the application, it will throw a `java.lang.UnsupportedOperationException`.

A `TrayIcon` can be added to the `SystemTray`. It can have an image, a `tooltip(text)`, and it is also associated with a set of listeners. Various `MouseEvents`, like `MousePressed`, `MouseReleased`, `MouseClicked` are generated by a `TrayIcon` and provide the facility for notification of the events. Some of the events are processed by `TrayIcon` itself. Suppose, a user performs right-click on the `TrayIcon`, then the `TrayIcon` displays the pop-up menu. An `ActionEvent` is also generated by `TrayIcon`.

Here's the inheritance diagram for the `SystemTray` class:

```
java.lang.Object
  |____ java.awt.SystemTray
```

You can find the methods of the `SystemTray` in Table 6.6.

Table 6.6: Methods of the SystemTray class

Method	Does this
`void add(TrayIcon)`	It is used for adding a `TrayIcon` to the `SystemTray`
`void addPropertyChangeListener(String propertyName, PropertyChangeListener listener)`	It adds a `PropertyChangeListener` for the indicated property
`PropertyChangeListener[] getPropertyChangeListeners()`	It returns an array of all listeners that are associated with the named property
`static SystemTray getSystemTray()`	It returns an `SystemTray` instance that represents the desktop tray's area
`TrayIcon[] getTrayIcons()`	It returns the array of all icons added to the tray by this application
`Dimension get TrayIconSize()`	It returns the size in pixels, of the space that a tray icon will occupy in the system tray
`static boolean isSupported()`	It returns whether the `SystemTray` is supported on the current platform
`void remove (TrayIcon trayicon)`	It removes the specified `TrayIcon` from the `SystemTray`
`void removePropertyChangeListener(String propertyName, PropertyChangeListener listener)`	It removes a `PropertyChangeListener` from the listener list for the indicated property

Let's look at an example of `SystemTray`. Here, the `SysTrayDemo` class has used `SystemTray` and `TrayIcon`. You can get the instance of `SystemTray` by calling the method `getSystemTray()`. The `SystemTray` class represents the tray bar, and each icon on it is represented by the `TrayIcon` class. A `TrayIcon` can be added to the `SystemTray` by invoking `SystemTray.add()` method. The `SysTrayDemo` class adds a `TrayIcon` with an image, a tooltip and a pop-up menu.

```
import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JOptionPane;
import javax.swing.SwingUtilities;
public class SysTrayDemo {
  private static void createGUI() {
        if(!SystemTray.isSupported()) {
                System.out.println("SystemTray is not supported");
                return; }
        SystemTray systray=SystemTray.getSystemTray();
        Toolkit toolkit = Toolkit.getDefaultToolkit();
        Image image = toolkit.getImage("image.jpg");
        PopupMenu popmenu = new PopupMenu();//Menu item to show the message
        MenuItem msgItem= new MenuItem("Show Message");
        msgItem.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                        JOptionPane.showMessageDialog(null,"System Tray
                        Demo"); } });
        popmenu.add(msgItem); //create menu item to close application
        MenuItem closeitem = new MenuItem("close");
        closeitem.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                        System.exit(0); }});
        popmenu.add(closeitem);
        TrayIcon icon = new TrayIcon(image,"SystemTray Demo",popmenu);
        icon.setImageAutoSize(true);
```

```
            try { systray.add(icon); }
            catch (AWTException e) {
                    System.out.println("Tray icon cannot be added to system
                    tray"); }
            }
      public static void main(String args[]) {
            SwingUtilities.invokeLater(new Runnable() {
                    public void run(){ createGUI(); }
            });
      }
   }
```

When you run the `SysTrayDemo` class, you'll find the icon of the `SystemTray` in the Taskbar Status Area. If you right-click on the icon, you'll get the menu. The output of the code is as shown in Figure 6.18:

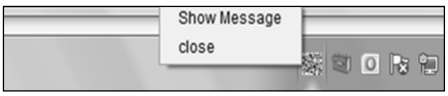


Figure 6.18: System Tray Demo after right click

When you'll select the "Show Message," you get the output as shown in Figure 6.19:

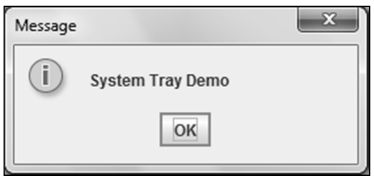


Figure 6.19: Message after selecting "Show Message"

Splash Screen

Splash screen informs user that the application is starting up. It is a standard part of any GUI application. A polished Splash screen of the application can be used for presenting marketing information and legal information, such as copyright information, third party logo, and so on. To see the Splash screen, a user can be sure that the application is starting.

For creating the Splash screen in Java application, you can use AWT or Java Foundation Classes/Swing (JFC/Swing). There should be a minimum delay between the application's startup and the pop up of the Splash screen. The application is permitted to display the Splash screen much before the starting of Java Virtual Machine. It is possible to decode the image and then display it in a simple window by an application launcher. The Splash screen is also capable of displaying any image format like `jpeg`, `gif`, `png` with transparency and animation. You can use one of the two ways to show the native Splash screen:

If you are using the command-line for running the application, then to show the native Splash screen, use the `-splash:` Java application launcher option. Here's an example:

```
Java -splash:filename.png HelloSplash
```

If application is packaged in a `.jar` file, a Splash screen can be displayed by using the `SplashScreen-Image` option in a manifest file. You need to save the image in a `jar` archive and mention the path in the option. Suppose, there is a manifest, file `manif.mf` Now use this code in the manifest file:

```
Manifest-Version: 1.0
Main-class: HelloSplash
SplashScreen-Image: filename.png
```

Among these two ways, the command-line interface has precedence over the manifest setting.

Here's the inheritance diagram for the `SplashScreen` class:

```
java.lang.Object
   |____ java.awt.SplashScreen
```

You can find the methods of the `SplashScreen` class in Table 6.7.

Table 6.7: Methods of the SplashScreen class

Method	Does this
`void close()`	It is used for hiding the Splash screen, closing the window, and releasing all associated resources
`Graphics2D createGraphics()`	It creates a graphics context for the Splash screen overlay image, which allows you to draw over the Splash screen
`Rectangle getBounds()`	It returns the bounds of the Splash screen window as a `Rectangle`
`URL getImageURL()`	It returns the current Splash screen image
`Dimension getSize()`	It returns the size of the Splash screen window as a `Dimension`
`static SplashScreen getSplashScreen()`	It returns the `SplashScreen` object used for Java startup Splash screen control on systems supporting dislpay
`boolean isVisible()`	It determines whether the Splash screen is visible or not
`void setImageURL (URL imageURL)`	It changes the Splash screen image
`void update()`	It updates the Splash window with current contents of the overlay image

Let's look at an example. Here, the `SplashScreenDemo` class is used to display a Splash screen before the JVM starts. The instance of the `SplashScreen` class is obtained by calling the static `getSplashScreen()` method, and the `createGraphics()` method returns a Graphics 2D object that allows you to draw over the Splash screen:

```
 import java.awt.*;
import java.awt.SplashScreen;
import javax.swing.*;
public class SplashScreenDemo {
  private static void createImage() {
        SplashScreen splashscreen = SplashScreen.getSplashScreen();
        if(splashscreen!=null){
                Graphics2D g = (Graphics2D) splashscreen.createGraphics();
        for(int i=0;i<10;i++){
                String message = "Processing"+ i+ "of 10....";
                g.setComposite(AlphaComposite.Clear);
        g.fillRect(130,350,280,40);
        g.setPaintMode();
        g.setColor(Color.RED);
        g.drawString(message,130,360);
        g.fillRect(130,370,i*30,20);
        splashscreen.update();
        try {
                Thread.sleep(500);
                }catch(InterruptedException e) { e.printStackTrace(); }
        }
  }
  JFrame frame = new JFrame("Splash Screen demo");
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setSize(300,200);
  frame.setLocationRelativeTo(null);
  frame.setVisible(true); }
  public static void main (String args[]) {
        SwingUtilities.invokeLater(new Runnable(){
                public void run(){ createImage(); }
        });
  }
}
```

Before running the `SplashScreenDemo` class, you have to make sure that the `Java8.jpg` file is in the same directory. Run the class as follows:

```
java –splash:Java8.jpg  SplashScreenDemo
```

When you run the `SplashScreenDemo` class, you'll find that there is nothing more than a blank frame. However, before it is uploaded, you'll see a Splash screen like that in Figure 6.20:



Figure 6.20: Image found after excuting "c:\>java –splash:Java8.jpg SplashScreenDemo"

Dialog Modality

A dialog box comprises a title and a border. It is used to take input from user and represented as a top-level pop-up window. An application uses a modal dialog box for blocking the input to some other top-level windows. The modal dialog box captures the window focus until it gets closed.

In JDK 5.0 and earlier, there was a problem with the modality model. When an application shows any modal dialog box, the dialog box blocks the user from interacting with the `JavaHelp` tool windows (an API to display help information). This problem has been resolved in Java SE 7 using a new AWT modality model. This new modality model is helpful for the developer. It provides the developer to scope or limit the dialog modality, blocking according to the selection of the modality type by the developer.

There are four modality types:

- Modeless
- Document modal
- Application modal
- Toolkit modal

Modeless

It does not block any other window until it is visible.

Document Modal

In this modal, all windows are blocked from the same document, except those from its child hierarchy.

Application Modal

In this modal, all windows are blocked from the same application, except those from its child hierarchy.

Toolkit Modal

All windows are blocked that run in the same toolkit, except from its child hierarchy. A dialog box will use the application modal by default, whereas with JDKs, prior to Java SE 6, it is modeless by default.

Gif Writer

Sometimes, Graphics Interchange Format (GIF) is also referred to as Graphics Interchange File Format (GIFF). It is well known that there are three graphic file formats:

- Joint Photographic Experts Group (JPEG)
- GIF
- PNG (Portable Network Graphics)

JPEG supports a compression technique that can be lost, i.e., the images cannot be restored to their original form if any images are compressed in the JPEG format, while GIF supports lossless compression but can only render 256 colors. These images are basically for simple images, such as buttons, labels, and logos. JPEG is used for complex graphics, like photos. PNG is also superior to GIF, but it is not as widespread as GIF. Therefore, GIF is used everywhere on the World Wide Web.

The `ImageConverter` class converts any image format to a GIF:

```
import java.awt.image.BufferedImage;
import javax.imageio.*;
import java.io.*;
import java.util.*;
public class ImageConverter
{
  public static void main(String args[]) throws IOException
  {
        System.out.println("Supported conversion Formats:");
        System.out.println(Arrays.toString(ImageIO.getWriterFormat
        Names()));
        if(args.length == 0)
        {
                System.err.println("Missing Input Filename");
                System.exit(-1);
        }
        String filename = args[0]; File inputFile = new File(filename);
        BufferedImage input = ImageIO.read(inputFile);
        File outputFile = new File(filename+ ".gif");
        ImageIO.write(input, "GIF", outputFile);
  }
}
```

Text Antialiasing

Text antialiasing is used for reducing the jaggedness of text or images onto screens with discrete pixel values. When we draw a diagonal line on the monitor, we turn pixels on and off. After looking it closely, you'll find that stair-step effect. Mustang has provided the facility for supporting the LCD displays by using the subpixel text to increase the text resolutions. Each color element of a pixel is called subpixel. These subpixels rendering is based on the phenomenon that modifying the intensities of subpixels can cause the location of pixels to shift, without changing the color.

The following five fields exist in the `java.awt.RenderingHints` class:

- VALUE_TEXT_ANTIALIAS_LCD_HRGB
- VALUE_TEXT_ANTIALIAS_LCD_HBGR
- VALUE_TEXT_ANTIALIAS_LCD_VRGB
- VALUE_TEXT_ANTIALIAS_LCD_VBGR
- VALUE_TEXT_ANTIALIAS_GASP

You can use each of these values when using the `setRenderingHint()` method of the Graphics 2D object to set the key text antialiasing value.

Summary

In this chapter, we have discussed about Abstract Window Toolkit, its applications, and event handling. You have learned how to use the Abstract Window Toolkit and creation of applets. You have also read about handling non- Java browsers and the use of init(), start(), stop(), destroy(), paint(), and update() methods. The chapter has also helped you to draw the graphics and paramters in applets and add controls to them. Moreover, the adapter classes and the anonymous inner adapter classes have been discussed in the chapter. Toward the end, you learn to run the applications in applets and set applets security policies.

In the next chapter, we'll learn about text fields, buttons, checkboxes, radio buttons, and various layouts of the AWT component class.



7

AWT—Text Fields, Buttons, Checkboxes, Radio Buttons, and Layouts

If you need an immediate solution to:	*See page:*
Using Text Fields	218
Using Labels	220
Using Buttons	221
Using Checkboxes	224
Using Radio Buttons	227
Layout Managers	228
Flow Layouts	228
Grid Layouts	231
Using Panels	232
Border Layouts	234
Card Layouts	236
Grid Bag Layouts	238
Using Insets and Padding	243
Creating Your Own Layout Manager	244

In Depth

This chapter is all about a number of important AWT components—text fields, buttons, checkboxes, and radio buttons. Now that we are dealing with visible components, we'll also take a look at Java layout managers, which let you arrange the components in an applet or application, as you like. We'll also take a look at Java panels, which let you assemble components together on one surface and submit the resulting surface to a layout manager. We'll start by taking a look at text fields.

Text Fields

Text fields are the basic text-handling components of the AWT. These components handle a one-dimensional string of text; they let you display text, write passwords by masking typed text, enter and read the text the user has entered, and more. In addition to buttons, these components are the most fundamental AWT components.

Buttons

Buttons provide users with a quick way to start some action—all they have to do is click them. Every user is familiar with buttons, and we have already taken a look behind the scenes on how buttons work in code when discussing event handling in the previous chapter. You can give buttons a caption, such as "Display Text". When the user does click the button, your code is notified if you have registered to handle events from the button.

Checkboxes

Checkboxes are much like buttons, except that they are *dual state*, which means they can appear as selected or unselected. When selected, they display a visual indication of some kind, such as a checkmark or an X (it varies by operating system in AWT programming, which is one of the reasons Sun introduced Swing, which can display components with the same look across many operating systems). The user can click a checkbox to select an option of some kind (such as the items on a sandwich), to enable automatic spell checking, or to enable printing while he or she is doing something else. You use checkboxes to let the user select non-exclusive options; for example, both automatic spell checking and background printing may be enabled at the same time. Radio buttons, however, are a different story.

Radio Buttons

You let the user select one of a set of mutually exclusive options by using radio buttons. Only one of a set of option buttons can be selected at one time; for example, using radio buttons, you can let the user select a printing color or the day of the week. In AWT, radio buttons are actually a type of checkbox; and when selected, they display a round dot or a clicked square or some other indication (again, the visual indication depends on the operating system). You use radio buttons in groups, and you'll see how that works in this chapter.

Layouts

So far, we have just added components to applets and applications by using the `add` method. This method is actually a method of the default layout manager—the *flow layout manager*. This layout manager is responsible for arranging components in AWT applets, by default. The flow layout manager arranges components much like a word processor might arrange words—across the page and then wrapped to the next line as needed, creating what Oracle calls a *flow of components*. You'll see that you can customize flow layouts to some extent, such as left-, center-, or right-aligning components. However, the limitations of the flow layout are clear, especially if you are counting on components maintaining some position with respect to the others, because if the user resizes your applet or application, the components will all move around. On the other hand, there are other AWT layout managers (and quite a few new ones in Swing), and we'll cover the AWT grid, border, card, and grid bag layout managers in this chapter.

Why can't you just set where a component goes and forget it? Novice Java programmers are often frustrated by having to deal with AWT layout managers and want to know why they can't just give the coordinates for the components they want to use and be done with it. In fact, you can, although you are responsible for handling the

case where windows are resized and components should be moved. To position components where you want them, you can indicate that you want no layout manager at all. Then you can size and locate the components, as you want them, using `add() method` to display them, like this:

```
setLayout(null);
text1 = new TextField(20);
text1.setSize(200,50);
text1.setLocation(20,20);
add(text1);
```

This adds a text field of size (200,50) at location (20,20) in a container, such as an applet window. Therefore, as you can see, you can add components to containers without any layout manager at all, something that's useful to bear in mind if the AWT layouts frustrate you too much. One very useful AWT container to use with layouts is the Panel component. You can arrange components in a panel and then add the panel, itself, to the layout of an applet or application. You'll see how to do that in this chapter. That's it. Now that we have reviewed what's in this chapter, it's time to turn to the "Immediate Solutions" section.

Immediate Solutions

Using Text Fields

"Hey," says the Novice Programmer, "I want to let users type in a password, but that silly Johnson keeps standing over peoples' shoulders and watching as they type their passwords." "That's easily fixed," you say. "Just set the echo character of the text field to an asterisk or some similar character. Problem solved!"

You first saw and used text fields in the previous chapter; these components can display a single line of text, and the user can edit that text. Here's the inheritance diagram for the text field class, `TextField`:

```
java.lang.Object
  |____java.awt.Component
       |____java.awt.TextComponent
            |____java.awt.TextField
```

You can find the constructors of the `TextField` class in Table 7.1 and its methods in Table 7.2.

Table 7.1: Constructors of the TextField class

Constructor	Does this
`TextField()`	It constructs a new text field
`TextField(int columns)`	It constructs a new, empty text field with the indicated number of columns
`TextField(String text)`	It constructs a new text field with the indicated text
`TextField(String text, int columns)`	It constructs a new text field initialized with the indicated text and with the indicated number of columns

Table 7.2: Methods of the TextField class

Method	Does this
`void addActionListener (ActionListener l)`	It adds the indicated action listener to receive action events
`void addNotify()`	It creates the `TextField` object's peer
`boolean echoCharIsSet()`	It indicates whether the text field has a character set for echoing
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this `TextField`
`ActionListener[] getActionListeners()`	It returns an array of all the action listeners registered on this `TextField`
`int getColumns()`	It gets the number of columns in the text field
`char getEchoChar()`	It gets the character to be used for echoing
`<T extends EventListener>T[]getListeners(Class<T> listenerType)`	It returns an array of all the listeners currently registered as `FooListeners` on this `TextField`
`Dimension getMinimumSize()`	It gets the minimum dimensions for the text field
`Dimension getMinimumSize (int columns)`	It gets the minimum dimensions for a text field with the indicated number of columns
`Dimension getPreferredSize()`	It gets the preferred size of the text field
`Dimension getPreferredSize (int columns)`	It gets the preferred size of the text field with the indicated number of columns
`Dimension minimumSize()`	It is deprecated and replaced by `getMinimumSize()`
`Dimension minimumSize (int columns)`	It is deprecated and replaced by `getMinimumSize(int)`
`protected String paramString()`	It gets the parameter string representing the state of the text field

Table 7.2: Methods of the TextField class	
Method	**Does this**
`Dimension preferredSize()`	It is deprecated and replaced by `getPreferredSize()`
`Dimension preferredSize (int columns)`	It is deprecated and replaced by `getPreferredSize(int)`
`protected void processActionEvent(ActionEvent e)`	It processes events occurring in the text field by sending them to `ActionListener` objects
`protected void processEvent (AWTEvent e)`	It processes events on the text field
`void removeActionListener (ActionListener l)`	It removes the indicated action listener so that it no longer receives action events
`void setColumns (int columns)`	It sets the number of columns in the text field
`void setEchoChar(char c)`	It sets the echo character for the text field
`void setEchoCharacter (char c)`	It is deprecated and replaced by `setEchoChar(char)`
`void setText(String t)`	It sets the text in this text field to the indicated text

Let's look at an example. In this case, we'll create a password text field that will display an asterisk (`*`) each time the user types a character. You may be wondering how you can read the typed password in the text field. The answer is by using the text field's `getText()` method, which the text field inherits from the `Component` class. The typed password is displayed in the applet's status bar when the user presses the Enter key. Here's the code:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
        CODE=PasswordDemo.class
        WIDTH=250
        HEIGHT=200 >
</APPLET> */
public class PasswordDemo extends Applet implements ActionListener {
public TextField text1;
public void init() {
text1 = new TextField(30);
add(text1);
. . .
```

Next, we set the echo character in `text1` (the password text field) to `*` and add an action listener to that text field:

```
public void init()
{
        text1 = new TextField(30);
        add(text1);
  text1.setEchoChar('*');
  text1.addActionListener(this);
}
```

When the user presses Enter, the `actionPerformed()` method is called, so we override that method to set the text in the status bar of the applet using the `showStatus()` method:

```
public void actionPerformed(ActionEvent e)
{
        if(e.getSource() == text1) { showStatus(text1.getText()); }
  }
}
```

The result of this code appears in Figure 7.1. When the user types a password in the top text field and presses Enter, that password appears in the applet's status bar (not exactly what you'd call high security).

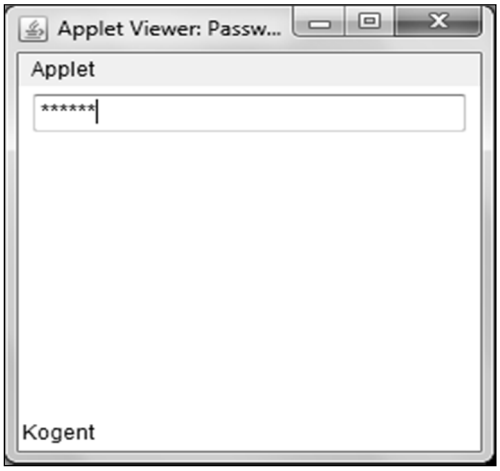


Figure 7.1: Reading a password in a text field

TIP

Another useful text field method is getSelectedText(), which lets you get the selected text in a text field.

You can find the process of Using Text Areas in detail in Chapter 8.

Using Labels

AWT labels are much like AWT text fields, except that the user can't edit the text in them. You can use labels to present no editable text, or as the name implies, to *label* other components. Here's the inheritance diagram for the `Label` class:

```
java.lang.Object
   |____java.awt.Component
          |____java.awt.Label
```

You can find the `Label` class's constructors in Table 7.3 and its methods in Table 7.4.

Table 7.3: Constructors of the Label class

Constructor	Does this
`Label()`	It constructs an empty label
`Label(String text)`	It constructs a new label with the specified string of text, left justified
`Label(String text, int alignment)`	It constructs a new label that presents the specified string of text with the specified alignment

Table 7.4: Methods of the Label class

Method	Does this
`void addNotify()`	It creates the peer for this label
`AccessibleContext getAccessibleContext ()`	It returns the `AccessibleContext` associated with this label
`int getAlignment()`	It gets the current alignment of this label
`String getText()`	It gets the text of this label
`protected String paramString()`	It returns the parameter string representing the state of this label
`void setAlignment(int alignment)`	It sets the alignment for this label to the specified alignment
`void setText(String text)`	It sets the text for this label to the specified text

You can justify the text in a label by passing the fields `Label.LEFT`, `Label.CENTER`, and `Label.RIGHT` to the label's constructor. Here's an example that creates three labels with the various possible text justifications:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
        CODE=LabelDemo.class
        WIDTH=200
        HEIGHT=200 >
</APPLET> */
public class LabelDemo extends Applet {
Label label1;
Label label2;
Label label3;
public void init() {
        label1 = new Label("Kogent Learning Solutions", Label.LEFT);
        add(label1);
        label2 = new Label("Kogent Learning Solutions", Label.CENTER);
        add(label2);
        label3 = new Label("Kogent Learning Solutions", Label.RIGHT);
        add(label3); }
}
```

Figure 7.2 shows the result of this applet.

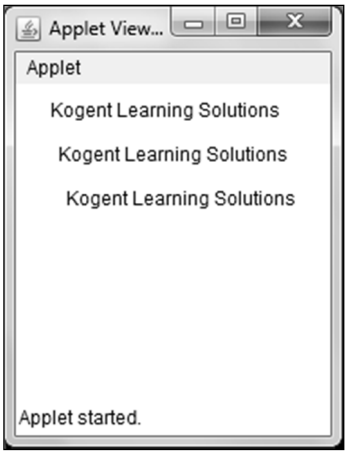


Figure 7.2: Justifying text in a label

Using Buttons

"I want to let users interact with my program," says the Novice Programmer. "I want to let them indicate what they want to do with just a click of the mouse, I want them to be able to select an action quickly and easily, and I want...." "Buttons," you say. "What you want are buttons." "Right," replies the NP.

Every GUI user is familiar with buttons—those elementary controls you click to signal to a program to start some action. For example, you might let the user click a button to change the background color of an application. Buttons are supported in the `java.awt.Button` class. Here's the lineage of that class:

```
java.lang.Object
   |____java.awt.Component
          |____java.awt.Button
```

You'll find the constructors for the `Button` class in Table 7.5 and the methods of this class in Table 7.6.

Table 7.5: Constructors of the Button class

Constructor	Does this
`Button()`	It constructs a button without any label
`Button(String label)`	It constructs a button with the indicated label

Table 7.6: Methods of the Button class

Method	Does this
`void addActionListener (ActionListener l)`	It adds the indicated action listener to receive action events from the button
`void addNotify()`	It creates the peer of the button
`AccessibleContext getAccessibleContext ()`	It gets the `AccessibleContext` associated with this `Button`
`String getActionCommand()`	It gets the command name of the event caused by the button
`ActionListener[] getActionListeners ()`	It gets an array of all the action listeners registered on this button
`String getLabel()`	It gets the label of the button
`<T extends EventListener> getListeners (Class<T> listenerType)`	It returns an array of all the listeners that are currently registered as `FooListeners` upon this `Button`
`protected String paramString()`	It gets the parameter string representing the state of the button
`protected void processActionEvent (ActionEvent e)`	It processes action events occurring on the button by sending them to any registered `ActionListener` objects
`protected void processEvent (AWTEvent e)`	It processes events in the button
`void removeActionListener (ActionListener l)`	It removes the action listener so that it no longer receives events from the button
`void setActionCommand(String command)`	It sets the command name for the action event caused by the button
`void setLabel(String label)`	It sets the button's label to the indicated string

To handle button events, you use the `ActionListener` interface, as you saw in the previous chapter. This interface has only one method, `actionPerformed()`, which is passed an object of the `ActionEvent` class when the button is clicked:

```
void actionPerformed(ActionEvent e) { . . . }
```

Here's the inheritance diagram of the `ActionEvent` class:

```
java.lang.Object
 |____java.util.EventObject
   |____java.awt.AWTEvent
         |____java.awt.event.ActionEvent
```

You'll find the methods of the `ActionEvent` class in Table 7.7.

Table 7.7: Methods of the ActionEvent class

Method	Does this
`String getActionCommand()`	It gets the command string associated with this action
`int getModifiers()`	It gets the modifier keys held down during the event
`long getWhen()`	It gets the timestamp of when this event occurred
`String paramString()`	It gets a parameter string identifying the event

As you saw in the previous chapter, there are two main ways to determine which button was selected—using the `getSource()` method and using commands. We'll take a look at the `getSource()` way of doing things

first. Here's an example with a button that, when clicked, displays the message "Kogent Learning Solutions" in a text field (note that we register an action listener with the button and find out which button was clicked with the `getSource()` method before setting the appropriate text in the text field):

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class ButtonDemo1 extends Applet implements ActionListener {
/* <APPLET
   CODE=ButtonDemo1.class
   WIDTH=200
   HEIGHT=200 >
</APPLET> */
   TextField text1;
   Button button1;
         public void init() {
   text1 = new TextField(20);
   add(text1);
   button1 = new Button("Click Me!");
   add(button1);
         button1.addActionListener(this); }
   public void actionPerformed(ActionEvent e) {
   String text = new String ("Kogent Learning Solutions");
   if(e.getSource() == button1) { text1.setText(text); }
   }
}
```

The result of this code appears in Figure 7.3.



Figure 7.3: Handling button clicks

You can also use commands with buttons; in the previous chapter, you saw that the default command associated with a button is its caption. However, when captions change during program execution (for example, by using the `setLabel()` method), you should associate a command with the button yourself. You can give a button a command string with the `setActionCommand()` method like this:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
   CODE=ButtonDemo2.class
   WIDTH=200
   HEIGHT=200>
</APPLET> */
public class ButtonDemo2 extends Applet implements ActionListener
{
   TextField text1;
   Button button1;
   public void init() { text1 = new TextField(20);
   add(text1); button1 = new Button("Click Me!");add(button1);
   button1.setActionCommand("button1");
   button1.addActionListener(this);
}
```

Now in the `actionPerformed()` method, you can get the command for the button that was clicked with the `getActionCommand()` method. If the button that was clicked is the right one, you can place the message in the text field, like this:

```
public void actionPerformed(ActionEvent e) {
String text= new String ("Kogent Learning Solutions");
String cmd = e.getActionCommand();
if(cmd.equals("button1")) { text1.setText(text); }
  }
}
```

Using Checkboxes

"Now I have another problem," the Novice Programmer says. "I want to let users select what they want on a pizza, so I'd like a button that, when clicked, stays clicked so that the users know what they have already selected." "No problem at all," you say. "Don't use buttons." "No?" the NP asks. "No," you say. "Use checkboxes instead."

A checkbox allows the user to select options; when the user clicks a checkbox, a visual indication of some kind, such as a checkmark (the indicator varies by operating system when using the AWT), is used to specify that the option is selected. Clicking the checkbox again deselects the checkbox. In AWT, checkboxes are supported with the `java.awt.Checkbox` class, which has the following inheritance diagram:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Checkbox
```

You can find the constructors for the `Checkbox` class in Table 7.8 and the methods of this class in Table 7.9. You can use the `setState()` and `getState()` methods to set and get the state of a checkbox, respectively.

Table 7.8: Constructors of the Checkbox class

Constructor	Does this
`Checkbox()`	It creates a checkbox without a label
`Checkbox(String label)`	It creates a checkbox with the indicated label
`Checkbox(String label, boolean state)`	It creates a checkbox with the indicated label and sets the indicated state
`Checkbox(String label, boolean state CheckboxGroup group)`	It creates a checkbox with the indicated label and sets it in the indicated state and checkbox group
`Checkbox(String label, CheckboxGroup group, boolean state)`	It constructs a checkbox with the indicated label in the indicated checkbox group and sets it to the indicated state

Table 7.9: Methods of the Checkbox class

Method	Does this
`void addItemListener (ItemListener l)`	It adds the indicated item listener to the checkbox
`void addNotify()`	It creates the peer of the checkbox
`AccessibleContext getAccessibleContext ()`	It gets the `AccessibleContext` associated with this `Checkbox`
`CheckboxGroup getCheckboxGroup()`	It gets the checkbox's group
`ItemListener[] getItemListeners()`	It returns an array of all item listeners registered on the checkbox
`String getLabel()`	It gets the label of the checkbox
`<T extends EventListener> T[] getListeners(Class<T> listenerType)`	It gets an array of all the objects currently registered as `FooListeners` upon this `Checkbox`
`Object[] getSelectedObjects()`	It gets an array (of length 1) that contains the checkbox label (or null if the checkbox is not selected)
`boolean getState()`	It determines whether the checkbox is selected or not

Table 7.9: Methods of the Checkbox class	
Method	**Does this**
`protected String paramString()`	It gets a parameter string representing the state of the checkbox
`protected void processEvent (AWTEvent e)`	It processes events on the checkbox
`protected void processItemEvent (ItemEvent e)`	It processes item events occurring on the checkbox by sending them to any registered `ItemListener` objects
`void removeItemListener (ItemListener l)`	It removes the indicated item listener so that the item listener no longer receives item events from the checkbox
`void setCheckboxGroup (CheckboxGroup g)`	It sets the checkbox's group to the given checkbox group
`void setLabel(String label)`	It sets the checkbox's label to the given string
`void setState (boolean state)`	It sets the state of the checkbox to the given state

Here's an example in which we add four checkboxes to an applet; and when the user clicks a checkbox, we indicate which checkbox was clicked in a text field. Note that checkboxes do not use the `ActionListener` interface as buttons do; instead, they use the `ItemListener` interface, which is set up to handle components that can be selected or deselected. The `ItemListener` interface has only one method, `itemStateChanged()`, which is passed as a parameter of the class `ItemEvent`:

```
void itemStateChanged(ItemEvent e)
```

You can find the methods of the `ItemEvent` class in Table 7.10.

Table 7.10: Methods of the ItemEvent class	
Method	**Does this**
`Object getItem()`	It gets the item affected by the event
`ItemSelectable getItemSelectable()`	It gets the originator of the event
`int getStateChange()`	It gets the type of state change (selected or deselected)
`String paramString()`	It gets a parameter string matching this event

Here's how we add the checkboxes to an applet and add an `ItemListener` interface to each one:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
        CODE=CheckboxDemo1.class
        WIDTH=200
        HEIGHT=200>
</APPLET> */
public class CheckboxDemo1 extends Applet implements ItemListener {
Checkbox check1, check2;
TextField text1;
public void init() {
check1 = new Checkbox("Pizza");
add(check1);
check1.addItemListener(this);
check2 = new Checkbox("Burger");
add(check2);
check2.addItemListener(this);
text1 = new TextField(20);
add(text1);
}
```

Now we override the `itemStateChanged()` method, determining which checkbox was clicked by using the `ItemEvent` object's `getItemSelectable()` method:

```
public void itemStateChanged(ItemEvent e) {
  if(e.getItemSelectable() == check1) {
  text1.setText("You clicked Pizza"); }
```

```
    else if(e.getItemSelectable() == check2){
      text1.setText("You clicked Burger"); }
      }
    }
```

Note that you can also use the `ItemEvent` object's `getStateChanged()` method to determine whether a checkbox is selected or deselected. This method returns `Checkbox.SELECTED` or `Checkbox.DESELECTED`. And, of course, you can use the checkbox's `getState()` method to make the same determination. You can also set the state of the checkbox with the `setState()` method. The result of this applet appears in Figure 7.4.



Figure 7.4: Handling checkbox clicks

Because it's tedious to have many `if` statements in the `if-else` ladder as done in the preceding code, we can just display the checkbox that was clicked by getting its label directly, as shown in the following code:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
        CODE=CheckboxDemo2.class
        WIDTH=200
        HEIGHT=200 >
</APPLET> */
public class CheckboxDemo2 extends Applet implements ItemListener {
Checkbox check1, check2;
TextField text1;
public void init() {
check1 = new Checkbox("Pizza"); add(check1);
check1.addItemListener(this);
check2 = new Checkbox("Burger"); add(check2);
check2.addItemListener(this);
text1 = new TextField(20);
add(text1); }
  public void itemStateChanged(ItemEvent e)
  {
        text1.setText("You clicked " +
        ((Checkbox) e.getItemSelectable()).getLabel());
  }
}
```

This new version of the applet appears in `CheckboxDemo2.java`. There's another kind of checkbox you can use—*radio buttons*—and we'll take a look at them in the next solution.

Using Radio Buttons

"Uh-oh," says the Novice Programmer, "I have another problem. I put some checkboxes in my program, so the users could select the day of the week, but one user selected Wednesday *and* Friday." "Well," you say, "you should use radio buttons to display exclusive options, such as the day of the week, not checkboxes."

In AWT programming, radio buttons are a special kind of checkbox, and you use radio buttons in groups. Only one radio button in a group may be selected at one time; when the user selects one radio button in a group, the others in the group are automatically deselected.

When you add checkboxes to a checkbox *group*, they become radio buttons automatically. AWT supports checkbox groups with the `CheckboxGroup` class. This class has only one constructor, `CheckboxGroup`, which takes no parameters, and you'll find the methods of this class in Table 7.11. Note that because radio buttons are really checkboxes, you can use `Checkbox` methods with them, such as `getState()` and `setState()`.

Table 7.11: Methods of the CheckboxGroup class

Method	Does this
`Checkbox getCurrent()`	It is deprecated and replaced by `getSelectedCheckbox()`
`Checkbox getSelectedCheckbox()`	It gets the currently selected checkbox in this checkbox group
`void setCurrent (Checkbox box)`	It is deprecated and replaced by `setSelectedCheckbox(Checkbox box)`
`void setSelectedCheckbox (Checkbox box)`	It sets the currently selected checkbox in this group
`String toString()`	It returns a string representing the checkbox group, including the value of its current selection

Note, for example, that you can determine which radio button is selected in a group with the `CheckboxGroup` class's `getSelectedCheckbox()` method, and you can set which one is selected with the `setSelectedCheckbox()` method. Here's an example in which we create a checkbox group named `group`, and we add two radio buttons to that group. You add a radio button to a checkbox group by adding the group as a parameter in a checkbox's constructor, which turns the checkbox into a radio button. Here's what this looks like:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
        CODE=RadioButtonDemo.class
        WIDTH=200
        HEIGHT=200>
</APPLET> */
public class RadioButtonDemo extends Applet implements ItemListener {
CheckboxGroup group; Checkbox check1, check2;
TextField text1;
public void init() {
        group = new CheckboxGroup();
        check1 = new Checkbox("Male", false, group);
        add(check1);
        check1.addItemListener(this);
        check2 = new Checkbox("Female", false,group);
        add(check2);
        check2.addItemListener(this);
        text1 = new TextField(20);
        add(text1);
}
```

Note that we have added an `ItemListener` interface to each radio button, so we can implement the `ItemListener` interface and the `itemStateChanged()` method to indicate which radio button was clicked, like this:

```
 public void itemStateChanged(ItemEvent e) {
```

```
        text1.setText("You selected " +
        ((Checkbox) e.getItemSelectable()).getLabel()); }
        }
```

The result of this code appears in Figure 7.5.



Figure 7.5: Handling radio button clicks

Now that we are working with components in visual programs, it's time to give some consideration as to how those components are arranged. We'll do that in the next few solutions.

Layout Managers

A layout manager organizes the controls automatically in a window with the help of certain algorithms. It is an object which implements the `LayoutManager` interface. It also finds the size and position of the components in a container. Each object of the container is associated with a layout manager.

The `setLayout()` method is used for setting the layout manager. If there is no `setLayout()` defined, then the default layout manager is used. This method can be defined as follows:

```
    void setLayout(LayoutManager  layoutManager)
```

where `layoutManager` is referred to the desired layout manager.

If you pass null for `layoutManager`, then it disables the layout manager. You'll need to determine the size and the position of the component manually. For this, you have to use the method `setBounds()`, which is defined by the component.

Flow Layouts

"Uh-oh," says the Novice Programmer, "Java's gone all wacky again!" "What's it done this time?" you ask. The NP says, "Well, I'm creating a multiplying calculator, and I want all the text fields to be stacked vertically, but they keep rearranging themselves into rows." "That's because you are using a flow layout," you say.

Java has a number of AWT layout managers that handle how components are displayed in containers, and you can install those layout managers into those containers. By default, applets use the flow layout manager, which arranges components like text in a word processor, including wrapping them to the next line. The constructors of the `FlowLayout` class appear in Table 7.12 and its methods in Table 7.13.

Table 7.12: Constructors of the FlowLayout class

Constructor	Does this
`FlowLayout()`	It constructs a new `FlowLayout` with centered alignment and a default 5 pixels horizontal and vertical gaps
`FlowLayout(int align)`	It constructs a new `FlowLayout` with the given alignment and a default 5 pixels horizontal and vertical gaps
`FlowLayout(int align, int hgap, int vgap)`	It creates a new flow layout manager with the given alignment and the given horizontal and vertical gaps.

Table 7.13: Methods of the FlowLayout class

Method	Does this
`void addLayoutComponent (String name, Component comp)`	It adds the specified component to the layout
`int getAlignment()`	It gets the alignment for the layout
`boolean getAlignOnBaseline ()`	It returns True if components are to be vertically aligned along their baseline
`int getHgap()`	It gets the horizontal gap between the components and between the components and borders of the container
`int getVgap()`	It gets the vertical gap between the components and between the components and borders of the container
`void layoutContainer (Container target)`	It lays out the container
`Dimension minimumLayoutSize (Container target)`	It returns the minimum dimensions needed to lay out the components in the target container
`Dimension preferredLayoutSize (Container target)`	It returns the preferred dimensions for the layout given the components in the target container
`void removeLayoutComponent (Component comp)`	It removes the indicated component from the layout
`void setAlignOnBaseline (Boolean alignOnBaseline)`	It sets whether or not the components should be vertically aligned along their baseline
`void setAlignment (int align)`	It sets the alignment
`void setHgap(int hgap)`	It sets the horizontal gap between the components and between the components and borders of the container
`void setVgap(int vgap)`	It gets the vertical gap between the components and between the components and borders of the container
`String toString()`	It returns a string representation of this `FlowLayout` object and its values

You do have some control over how the components in a flow layout are arranged; you can specify their alignment by using these `FlowLayout` fields:

- **`CENTER`**—It specifies that each row of components must be centered.
- **`LEADING`**—It specifies that each row of components must be justified to the leading edge of the container.
- **`LEFT`**—It specifies that each row of components must be left-justified.
- **`RIGHT`**—It specifies that each row of components must be right-justified.
- **`TRAILING`**—It specifies that each row of components must be justified to the trailing edge of the container.

The default flow layout is to center each component in a row, but here's how we create a new flow layout manager that right-justifies components by using the `setLayout()` method:

```
import java.applet.Applet;
import java.awt.*;
/* <APPLET
        CODE=FlowLayoutDemo.class
        WIDTH=300
        HEIGHT=200 >
</APPLET> */
public class FlowLayoutDemo extends Applet {
Button button1, button2, button3;
public void init() {
setLayout(new FlowLayout(FlowLayout.RIGHT));
button1 = new Button("Kogent");
add(button1);
button2 = new Button("Learning");
add(button2);
button3 = new Button("Solutions");
add(button3);
}
}
```

After a few buttons are added to this new flow manager, you can see that they are indeed right-justified, as shown in Figure 7.6.

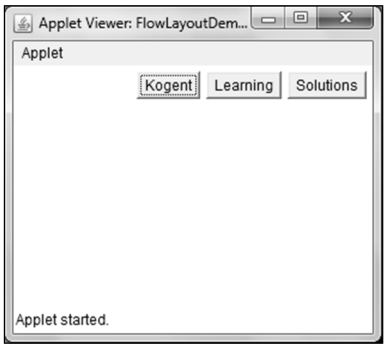


Figure 7.6: Right-justifying components in a flow layout

However, sometimes a flow layout is just not right. For example, take a look at this applet, which presents the multiplying calculator the Novice Programmer was trying to create:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
        CODE=Multiply.class
        WIDTH=290
        HEIGHT=200 >
</APPLET> */
public class Multiply extends Applet implements ActionListener {
        TextField num1, num2, result;
        Label multiply; Button button1;
        public void init() {
                setLayout(new FlowLayout(FlowLayout.CENTER));
                num1 = new TextField(10); add(num1);
                multiply = new Label("*"); add(multiply);
                num2 = new TextField(10); add(num2);
                button1 = new Button("="); add(button1);
                button1.addActionListener(this);
                result = new TextField(10);
        add(result); }
        public void actionPerformed(ActionEvent e) {
                if(e.getSource() == button1) {
                        int product = Integer.parseInt(num1.getText())
                        *Integer.parseInt(num2.getText());
                result.setText(String.valueOf(product)); }
        }
}
```

The result appears in Figure 7.7:

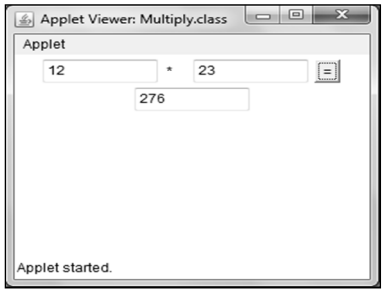


Figure 7.7: A first attempt at a multiplying calculator

As you can see in the figure, the text fields are arranged to suit the flow layout manager, and the result is hardly what the NP intended. To fix this situation, we'll take a look at another layout manager—the grid layout manager—next.

Grid Layouts

"So how do I fix my multiplying calculator? The text fields are all over the place," the NP asks. "You have to use a different layout manager," you say, "such as the grid layout manager."

The grid layout manager lets you add components to a container, positioning them in a grid. You tell the grid layout manager what the dimensions of your grid will be—such as five rows and four columns—and when you add components to the layout, they'll be added starting left of the first row. After the first row is filled, components are placed into the first column of the second row, and that row is then filled, and so on. Note that each component is given the same size and dimensions.

Here's the inheritance diagram for the grid layout manager:

```
java.lang.Object
  |____java.awt.GridLayout
```

You'll find the constructors for the `GridLayout` class in Table 7.14 and the methods of this class in Table 7.15.

Table 7.14: Constructors of the GridLayout class

Constructor	Does this
`GridLayout()`	It creates a grid layout with one column per component in one row
`GridLayout(int rows, int cols)`	It creates a grid layout with the given number of rows and columns
`GridLayout(int rows, int cols, int hgap, int vgap)`	It creates a grid layout with the given number of rows and columns as well as the given amount of spacing

Table 7.15: Methods of the GridLayout class

Method	Does this
`void addLayoutComponent (String name,Component comp)`	It adds the indicated component to the layout
`int getColumns()`	It gets the number of columns in the layout
`int getHgap()`	It gets the horizontal gap between components
`int getRows()`	It gets the number of rows in the layout
`int getVgap()`	It gets the vertical gap between components
`void layoutContainer (Container parent)`	It lays out the indicated container by using this grid layout
`Dimension minimumLayoutSize (Container parent)`	It determines the minimum size of the container by using this grid layout
`Dimension preferredLayoutSize (Container parent)`	It determines the preferred size of the container by using this grid layout
`void removeLayoutComponent (Component comp)`	It removes the indicated component from the layout
`void setColumns(int cols)`	It sets the number of columns in the layout to the indicated value
`void setHgap(int hgap)`	It sets the horizontal gap between components to the indicated value
`void setRows(int rows)`	It sets the number of rows in the layout to the indicated value
`void setVgap(int vgap)`	It sets the vertical gap between components to the indicated value
`String toString()`	It gets a string representation of this layout

Here's an example in which we create a grid of five rows and one column by passing those dimensions to the grid layout manager's constructor as `GridLayout(5,1)`. When we start adding components to this layout, they'll be stacked one on top of the other. Here's how that looks:

```
import java.applet.Applet;
import java.awt.*;
```

```
import java.awt.event.*;
/* <APPLET
        CODE=GridLayoutDemo.class
        WIDTH=250
        HEIGHT=200>
</APPLET> */
public class GridLayoutDemo extends Applet implements ActionListener {
TextField num1, num2, result;
        Label multiply; Button button1;
        public void init() {
                setLayout(new GridLayout(5,1));
                num1 = new TextField(); add(num1);
                multiply = new Label("*"); add(multiply);
                num2 = new TextField(); add(num2);
                button1 = new Button("="); add(button1);
                button1.addActionListener(this);
                result = new TextField();
        add(result); }
        public void actionPerformed(ActionEvent e) {
                if(e.getSource() == button1) {
                        int product = Integer.parseInt(num1.getText())
                        *Integer.parseInt(num2.getText());
                result.setText(String.valueOf(product)); }
        }
}
```

The result of this code appears in Figure 7.8:

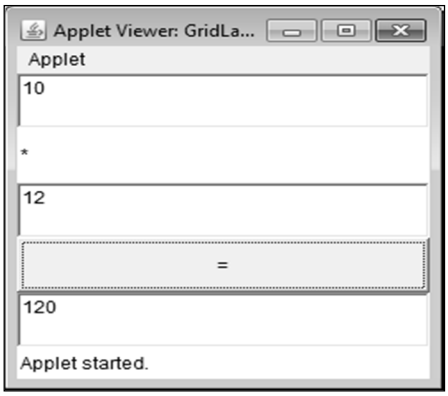


Figure 7.8: A second attempt of the multiplying calculator

As you can see in the figure, the components in this applet are stacked vertically, as intended.

One way of working with layouts is to divide your program's components into panels and to add those panels to the layout. This gives you more control over what controls go where. We'll take a look at panels in the next solution.

Using Panels

The NP is still dissatisfied with layout issues and says, "I need finer control in my layout—I have got a lot of components to display." "OK," you say. "You can use Java panels to get that kind of control." "Great! Tell me all about it," replies the NP.

The Java `Panel` class is a container you can use in your layouts. You add components to a panel and then add the panel to a layout. Because panels are containers, you can set their layout managers by using the `setLayout()` methods. Panels, themselves, are not visual containers—they don't have borders, for example—and exist only to contain components. Here's the inheritance diagram for the `Panel` class:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
                |____java.awt.Panel
```

The constructor for the `Panel` class appears in Table 7.16 and the method of this class appears in Table 7.17.

Table 7.16: The constructor of the Panel class

Constructor	Does this
`Panel()`	It creates a new panel with the default layout manager
`Panel(LayoutManager layout)`	It creates a new panel with the specified layout manager

Table 7.17: The method of the Panel class

Method	Does this
`void addNotify()`	It creates the panel's peer
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this panel

Here's an example in which we derive a new class `ButtonPanel` from the `Panel` class and add a few buttons to the panel in the `ButtonPanel` constructor:

```
import java.applet.Applet;
import java.awt.*;
/* <APPLET
        CODE=PanelDemo.class
        WIDTH=150
        HEIGHT=150 >
</APPLET> */
class ButtonPanel extends Panel {
Button button1, button2, button3, button4;
ButtonPanel() {
button1 = new Button("A");
add(button1);
button2 = new Button("B");
add(button2);
button3 = new Button("C");
add(button3);
button4 = new Button("D");
add(button4); }
}
```

Now we can arrange panels of this new class in a grid layout as follows:

```
public class PanelDemo extends Applet {
        ButtonPanel panel1, panel2, panel3, panel4;
        public void init() {
                setLayout(new GridLayout(2,2));
        panel1 = new ButtonPanel();
        add(panel1);
        panel2 = new ButtonPanel();
        add(panel2);
        panel3 = new ButtonPanel();
        add(panel3);
        panel4 = new ButtonPanel();
        add(panel4);
    }
 }
```

The result of this code appears in Figure 7.9:

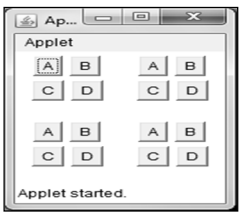


Figure 7.9: Arranging components using panels

As you can see in this figure, each panel has been added to the grid layout; in this way, using panels gives you finer layout control.

Border Layouts

The NP is back and asks, "Are there any more layouts I should know about?" "Of course," you say. "You should know the border layout, for example." The NP asks, "Does that arrange borders?" "Sort of," you say.

The border layout lets you arrange components around the border of a container, which is useful if you want to support, say, scrollbars and scroll a central component. Border layouts are the default in AWT windows, frame windows, and dialog boxes. You indicate where you want a component to go by passing text strings, such as "North", "East", and so on, to the `BorderLayout` constructor. Here's the inheritance diagram for `BorderLayout`:

```
java.lang.Object
  |____java.awt.BorderLayout
```

You can find the constructors of the `BorderLayout` class in Table 7.18 and its methods in Table 7.19.

Table 7.18: Constructors of the BorderLayout class

Constructor	Does this
`BorderLayout()`	It constructs a new border layout without any gap between components
`BorderLayout(int hgap, int vgap)`	It constructs a border layout with the indicated gaps between components

Table 7.19: Methods of the BorderLayout class

Method	Does this
`void addLayoutComponent (Component comp, Object constraints)`	It adds the component to the layout, using the given constraint object
`void addLayoutComponent (String name, Component comp)`	It is deprecated and replaced by `addLayoutComponent(Component, Object)`
`Object getConstraints (Component comp)`	It gets the constraints for the specified component
`int getHgap()`	It gets the horizontal gap between components
`float getLayoutAlignmentX (Container parent)`	It gets the alignment along the x-axis
`float getLayoutAlignmentY (Container parent)`	It gets the alignment along the y-axis
`Component getLayoutComponent(Container target, Object constraints)`	It returns the component that corresponds to the given constraint location based on the target `Container`'s component orientation
`Component getLayoutComponent (Object constraints)`	It gets the component that was added using the given constraint
`int getVgap()`	It gets the vertical gap between components
`void invalidateLayout (Container target)`	It invalidates the layout
`void layoutContainer (Container target)`	It lays out the container using this border layout
`Dimension maximumLayoutSize (Container target)`	It gets the maximum dimensions for the layout, given the components in the given target container
`Dimension minimumLayoutSize (Container target)`	It gets the minimum dimensions for the layout, given the components in the given target container
`Dimension preferredLayoutSize (Container target)`	It determines the preferred size of the target container using the layout manager, based on the components in the container

Table 7.19: Methods of the BorderLayout class	
Method	**Does this**
`void removeLayoutComponent (Component comp)`	It removes the given component from this border layout
`void setHgap(int hgap)`	It sets the horizontal gap between components
`void setVgap(int vgap)`	It sets the vertical gap between components
`String toString()`	It gets a string representation of the state of this border layout

Here's an example in which we create a panel containing a text field and add four buttons around the border of the panel. When the user clicks a button, the applet will indicate which button was clicked in the text field.

Here's how we create the panel that displays a text field:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
        CODE=BorderLayoutDemo.class
        WIDTH=200
        HEIGHT=200 >
</APPLET> */
class TextPanel extends Panel {
TextField text1;
TextPanel() { text1 = new TextField(20);
add(text1); }
}
```

Now we create a border layout, placing the text field panel in the center and adding four buttons around the border. You can specify where you want to place a component with strings, such as "North", "West", "Center" and so on. Therefore, creating the layout looks like this:

```
public class BorderLayoutDemo extends Applet implements ActionListener {
Button button1, button2, button3, button4;
TextPanel panel1;
public void init() {
setLayout(new BorderLayout());
button1 = new Button("1");
add("North", button1);
button1.addActionListener(this);
button2 = new Button("2");
add("West", button2);
button2.addActionListener(this);
button3 = new Button("3");
add("South", button3);
button3.addActionListener(this);
button4 = new Button("4");
add("East", button4);
button4.addActionListener(this);
panel1 = new TextPanel();
add("Center", panel1);
panel1.text1.setLocation(0, 0); }
public void actionPerformed(ActionEvent e) {
panel1.text1.setText("Button " +
((Button) e.getSource()).getLabel() + " clicked"); }
}
```

The result of this code appears in Figure 7.10:

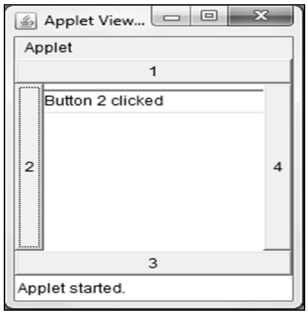


Figure 7.10: Creating a border layout

As you can see in this figure, the buttons appear around the perimeter of the central text field panel.

Card Layouts

"So," the NP says, "are there any other AWT layouts?" "There are," you say, "such as the card layout." "You are making that up," says the NP.

The card layout manager displays the containers you pass to it as *cards*. You assign each card a name, and then you move from one card to another using the show() method of `CardLayout` class. Here's the inheritance diagram for the `CardLayout` class:

```
java.lang.Object
   |____java.awt.CardLayout
```

Besides the `show()` method, you can also use the `first()`, `last()`, `next()`, and `previous()` methods of the `CardLayout` class to display specific cards. You'll find the `CardLayout` class's constructors in Table 7.20 and the methods of this class in Table 7.21:

Table 7.20: Constructors of the CardLayout class

Constructor	Does this
`CardLayout()`	It creates a new card layout
`CardLayout(int hgap, int vgap)`	It creates a new card layout with the given horizontal and vertical gaps

Table 7.21: Methods of the CardLayout class

Method	Does this
`void addLayoutComponent (Component comp, Object constraints)`	It adds the given component to the layout
`void addLayoutComponent (String name, Component comp)`	It gets deprecated and replaced by `addLayoutComponent(Component, Object)`
`void first (Container parent)`	It goes to the first card of the container
`int getHgap()`	It gets the horizontal gap between components
`float getLayoutAlignmentX (Container parent)`	It gets the alignment along the x-axis
`float getLayoutAlignmentY (Container parent)`	It gets the alignment along the y-axis
`int getVgap()`	It gets the vertical gap between components
`void invalidateLayout (Container target)`	It invalidates the layout, indicating that any information cached by the layout manager should be discarded

Table 7.21: Methods of the CardLayout class	
Method	**Does this**
`void last (Container parent)`	It goes to the last card of the container
`void layoutContainer (Container parent)`	It lays out the given container by using this card layout
`Dimension maximumLayoutSize (Container target)`	It gets the maximum dimensions for the layout, given the components in the given target container
`Dimension minimumLayoutSize (Container parent)`	It calculates the minimum size for the given panel
`void next(Container parent)`	It goes to the next card of the given container
`Dimension preferredLayoutSize (Container parent)`	It determines the preferred size of the container argument by using this card layout
`void previous(Container parent)`	It goes to the previous card of the given container
`void removeLayoutComponent (Component comp)`	It removes the given component from the layout
`void setHgap(int hgap)`	It sets the horizontal gap between components
`void setVgap(int vgap)`	It sets the vertical gap between components
`void show(Container parent, String name)`	It goes to the component that was added to the layout with the given name
`String toString()`	It gets a string representation of this card layout

Here's an example in which we create a class `CardLayoutDemo` that displays an applet containing a label. The user can click the applet to move to the next card and a label to indicate what text the present card label contains:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<APPLET
 CODE = CardLayoutDemo.class
 HEIGHT=150
 WIDTH=200>
</APPLET>*/
public class CardLayoutDemo extends Applet implements MouseListener
{
 Panel panel1, panel2, panel3, panel4;
 Label label1, label2, label3;
CardLayout cardLayout;
  public void init()
  {
        label1 = new Label("Kogent");
        label2 = new Label("Learning");
        label3 = new Label("Solutions");
        panel2 = new Panel();
        panel3 = new Panel();
        panel4 = new Panel();

        panel2.add(label1);
        panel3.add(label2);
        panel4.add(label3);

        panel1 = new Panel();
        panel1.add(panel2,"First");
        panel1.add(panel3,"Second");
        panel1.add(panel4, "Third");
        cardLayout = new CardLayout();
        panel1.setLayout(cardLayout);
        add(panel1);
```

```
            addMouseListener(this);
      . . .
    }
    }
```

In the main applet class, we create four panels and three labels. The layout of the panel `panel1` is set to card layout. The other three panels are added to the panel `panel1`, giving them the names "First", "Second", and "Third". We add mouse listener to the applet using the `addMouseListener()` method so that the user can move to the next card on clicking the applet. Next, we give body of the `mouseClicked()` method and call the `next()` method of the `CardLayout` class to move to the next card as follows:

```
    public void mouseClicked(MouseEvent e) {
            cardLayout.next(panel1); }
```

The other methods of the `MouseListener` interface are written without any code statements in the method body:

```
    public void mousePressed(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
```

The result appears in Figure 7.11:



Figure 7.11: Using a card layout

When the user clicks the applet, it moves to the next card in the layout, cycling through the cards over and over. That's what a card layout looks like and how it works. There's one more AWT layout to take a look at—grid bag layouts.

Grid Bag Layouts

"OK," says the NP. "I'm an expert with AWT layouts now." "No you are not," you say. "Not until you master the grid bag layout." The NP asks, "What the heck is a grid bag?"

Grid bag layouts are the most complex and most flexible of the AWT layouts, letting you specify more exactly than any of the other AWT layout managers where you want your components to go. Here's the inheritance diagram for the `GridBagLayout` class:

```
    java.lang.Object
      |____java.awt.GridBagLayout
```

You can find the constructor of the `GridBagLayout` class in Table 7.22 and its methods in Table 7.23.

Table 7.22: Constructor of the GridBagLayout class

Constructor	Does this
`GridBagLayout()`	It creates a grid bag layout manager

Table 7.23: Methods of the GridBagLayout class

Constructor	Does this
`void addLayoutComponent (Component comp, Object constraints)`	It adds the given component to the layout using the constraint object
`void addLayoutComponent (String name, Component comp)`	It has no effect as this layout manager doesn't use a per-component string
`protected void adjustForGravity (GridBagConstraints constraints, Rectangle r)`	It adjusts the x, y, width, and height fields to the right values according to the constraint geometry and pads
`protected void adjustForGravity (GridBagConstraints constraints, Rectangle r)`	It is obsolete and supplied only to maintain backward compatibility according to the arguments passed
`protected void arrangeGrid(Container parent)`	It arranges the grid in the `parent` container
`protected void arrangeGrid(Container parent)`	It is obsolete and supplied only to maintain backward compatibility according to the `parent` container
`GridBagConstraints getConstraints(Component comp)`	It gets the constraints for the given component
`float getLayoutAlignmentX (Container parent)`	It gets the alignment along the x-axis
`float getLayoutAlignmentY (Container parent)`	It gets the alignment along the y-axis
`int[][] getLayoutDimensions()`	It determines column widths and row heights for the layout grid
`protected GridBagLayoutInfo getLayoutInfo(Container parent, int sizeflag)`	It fills in an instance of `GridBagLayoutInfo` for the current set of manager children
`protected GridBagLayoutInfo GetLayoutInfo(Container parent, int sizeflag)`	It is obsolete and supplied only to maintain backward compatibility, `getLayoutInfo` should be preferred
`Point getLayoutOrigin()`	It determines the origin of the layout grid within the graphics space of the target container
`double[][] getLayoutWeights()`	It determines the weights of the layout grid's columns and rows
`protected Dimension getMinSize(Container parent, GridBagLayoutInfo info)`	It gets the minimum size of the container on the basis of information from `getLayoutInfo`
`protected Dimension getMinSize(Container parent, GridBagLayoutInfo info)`	It is obsolete and supplied only to maintain backward compatibility on the basis of the parent and grid baglayout
`void invalidateLayout (Container target)`	It invalidates the layout, indicating that any cached information should be discarded
`void layoutContainer (Container parent)`	It lays out the container using in grid bag layout
`Point location(int x, int y)`	It determines which cell in the layout grid contains the given point indicated by (x, y)
`protected GridBagConstraints lookupConstraints (Component comp)`	It retrieves the constraints for the given component
`Dimension maximumLayoutSize (Container target)`	It gets the maximum dimensions for the layout, given the components in the indicated target container

Table 7.23: Methods of the GridBagLayout class	
Constructor	**Does this**
`Dimension minimumLayoutSize (Container parent)`	It determines the minimum size of the container using this layout
`Dimension preferredLayoutSize (Container parent)`	It determines the preferred size of the container using this layout
`void removeLayoutComponent (Component comp)`	It removes the given component from the layout
`void setConstraints (Component comp, GridBagConstraints constraints)`	It sets the constraints for the given component in the layout
`String toString()`	It gets a string representation of this layout's values

Adding components to a container using a grid bag layout is something like using a grid layout, except you have more options, such as setting the relative widths and heights of the components. Setting up a grid bag layout is a two-step process: You first configure how you want a component to appear relative to the other components and then you add the component to the layout. You configure components using the `GridBagConstraints` class; the fields of that class appear in Table 7.24, its constructors in Table 7.25, and its method in Table 7.26.

Table 7.24: Fields of the GridBagConstraints class	
Field	**Does this**
`static int ABOVE_BASELINE`	It indicates the possible value for the `anchor` field
`static int ABOVE_BASELINE_LEADING`	It indicates the possible value for the `anchor` field
`static int ABOVE_BASELINE_TRAILING`	It indicates the possible value for the `anchor` field
`int anchor`	It is used when the component is smaller than its display area
`static int BASELINE`	It indicates the possible value for the `anchor` field
`static int BASELINE_LEADING`	It indicates the possible value for the `anchor` field
`static int BASELINE_TRAILING`	It indicates the possible value for the `anchor` field
`static int BELOW_BASELINE`	It indicates the possible value for the `anchor` field
`static int BELOW_BASELINE_LEADING`	It indicates the possible value for the `anchor` field
`static int BELOW_BASELINE_TRAILING`	It indicates the possible value for the `anchor` field
`static int BOTH`	It resizes the component both horizontally and vertically
`static int CENTER`	It puts the component in the center of its display area
`static int EAST`	It puts the component on the right side of its display area, centered vertically
`int fill`	It is used when the component's display area is larger than the component's requested size
`static int FIRST_LINE_END`	It places the component in the corner of its display area, where the first line would normally end for current `ComponentOrienation`
`static int FIRST_LINE_START`	It places the component in the corner of its display area, where the first line would normally start for current `ComponentOrienation`
`int gridheight`	It indicates the number of cells in a column for the component's display area
`int gridwidth`	It indicates the number of cells in a row for the component's display area
`int gridx`	It indicates the cell at the leading edge of the component's display area, where the first cell in the row has `gridx=0`
`int gridy`	It indicates the cell at the top of the component's display area, where the first cell in the row has `gridy=0`
`static int HORIZONTAL`	It resizes the component horizontally

Table 7.24: Fields of the GridBagConstraints class	
Field	**Does this**
`Insets insets`	It indicates the external padding of the component (the minimum amount of space between the components and edges of its display area)
`int ipadx`	It indicates the internal x padding of the component
`int ipady`	It indicates the internal y padding
`static int LAST_LINE_END`	It places the component in the corner of its display area, where the last line would normally end for current `ComponentOrienation`
`static int LAST_LINE_START`	It places the component in the corner of its display area, where the last line would normally start for current `ComponentOrienation`
`static int LINE_END`	It places the component centered along the edge of its display area, where the lines of text would normally end for current `ComponentOrienation`
`static int LINE_START`	It places the component centered along the edge of its display area, where the lines of text would normally start for current `ComponentOrienation`
`static int NONE`	It indicates not to resize the component
`static int NORTH`	It puts the component at the top of its display area, centered horizontally
`static int NORTHEAST`	It puts the component at the top-right corner of its display area
`static int NORTHWEST`	It puts the component at the top-left corner of its display area
`static int PAGE_END`	It places the component centered along the edge of its display area associated with the end of a page for the current `ComponentOrienation`
`static int PAGE_START`	It places the component centered along the edge of its display area associated with the start of a page for the current `ComponentOrienation`
`static int RELATIVE`	It indicates that this component is the next-to-last component in its column or row or that this component be placed next to the previously added component
`static int REMAINDER`	It specifies that this component is the last component in its column or row
`static int SOUTH`	It puts the component at the bottom of its display area, centered horizontally
`static int SOUTHEAST`	It puts the component at the bottom-right corner of its display area
`static int SOUTHWEST`	It puts the component at the bottom-left corner of its display area
`static int VERTICAL`	It resizes the component vertically
`double weightx`	It specifies how to distribute extra horizontal space
`double weighty`	It specifies how to distribute extra vertical space
`static int WEST`	It puts the component on the left side of its display area, centered vertically

Table 7.25: Constructors of the GridBagConstraints class	
Constructor	**Does this**
`GridBagConstraints()`	It creates a `GridBagConstraints` object
`GridBagConstraints(int gridx, int gridy, int gridwidth, int gridheight, double weightx, double weighty, int anchor, int fill, Insets insets, int ipadx, int ipady)`	It creates a `GridBagConstraints` object with all fields set to the passed values

Table 7.26: Method of the GridBagConstraints class	
Method	**Does this**
`Object clone()`	It creates a copy of this grid bag constraint

Let's look at an example of putting a grid bag layout to work. In this example, we'll create the applet you see in Figure 7.12; here, we have put three buttons into the top row of the layout, making the middle button twice as wide as the others, and we have added a text field underneath. When the user clicks a button, the color of the text field changes according to the name of color mentioned on the button, as you see in Figure 7.12:

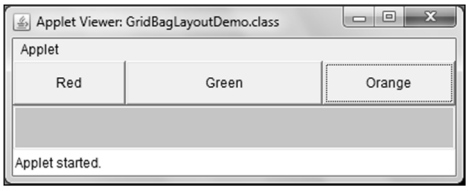


Figure 7.12: Using a grid bag layout

You can specify both x and y constraints for the components you add to a grid bag layout by using the `weightx` and `weighty` fields. These values represent the relative weights you want to give to the components in the same row or column. In this case, we'll give each component the same `y` weight, but give the second button twice the `x` weight as the other buttons.

We start by creating a grid bag layout and a constraints object:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
        CODE=GridBagLayoutDemo.class
        WIDTH=400
        HEIGHT=80 >
</APPLET> */
public class GridBagLayoutDemo extends Applet implements ActionListener {
Button button1, button2, button3;
TextField text1;
public void init() {
GridBagLayout gbl = new GridBagLayout();
GridBagConstraints cons = new GridBagConstraints();
setLayout(gbl);
constraints.weighty = 1;
constraints.fill = GridBagConstraints.BOTH;
. . .
```

Note that we are setting the constraints object's fill field to `GridBagConstraints.BOTH`, which means the layout manager will expand components in both dimensions to fill them out. Next, we create the first button and add it to the layout as follows:

```
public void init() {
GridBagLayout gbl = new GridBagLayout();
GridBagConstraints cons = new GridBagConstraints();
setLayout(gbl);
cons.weighty = 1;
cons.fill = GridBagConstraints.BOTH;
cons.weightx = 1;
button1 = new Button("Red");
gbl.setConstraints(button1, cons);
add(button1);
button1.addActionListener(this);
. . .
```

This first button has an `x` weight of 1, but we'll give the next button an `x` weight of 2 to make it twice as wide as the others in the same row:

```
public void init() {
GridBagLayout gbl = new GridBagLayout();
GridBagConstraints cons = new GridBagConstraints();
setLayout(gbl);
cons.weighty = 1;
cons.fill = GridBagConstraints.BOTH;
```

```
    cons.weightx = 1;
    button1 = new Button("Red");
    gbl.setConstraints(button1, cons);
    add(button1);
    button1.addActionListener(this);
    cons.weightx = 2;
    button2 = new Button("Green");
    gbl.setConstraints(button2, cons);
    add(button2);
    button2.addActionListener(this);
    cons.weightx = 1;
    button3 = new Button("Orange");
    cons.gridwidth = GridBagConstraints.REMAINDER;
    gbl.setConstraints(button3, cons);
    add(button3);
    button3.addActionListener(this);
    text1 = new TextField();
    cons.gridwidth = GridBagConstraints.REMAINDER;
    gbl.setConstraints(text1, cons);
    add(text1);
    }
```

All that's left is to handle button clicks, and we do that by displaying the number of the button that was clicked:

```
    public void actionPerformed(ActionEvent e) {
    if(e.getSource()==button1)
      text1.setBackground(Color.red);
    if(e.getSource()==button2)
      text1.setBackground(Color.green);
    if(e.getSource()==button3)
      text1.setBackground(Color.orange);
     }
    }
```

And that's all it takes. Now this code will create the applet you see in Figure 7.12.

Using Insets and Padding

"Say," says the NP, "is there any way of adding a border around the components in a layout?" "There sure is," you say. "You can use insets and padding."

Using insets, you can add space between the container edges and your components. You create insets with the `Insets` class, which has the following constructor:

```
    Insets (int top, int left, int bottom, int right)
```

Here's an example in which we add insets of 20 pixels on all edges to the applet developed in the previous solution:

```
    public Insets getInsets() {return new Insets(20,20,20,20); }
```

The result appears in Figure 7.13:

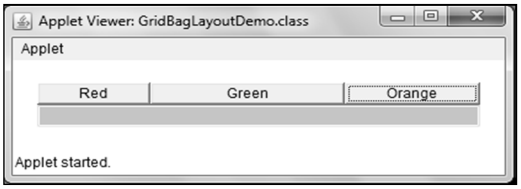


Figure 7.13: Using insets

As you can see in this figure, there's an inset of 20 pixels all around the components in the container. You can also pad individual components.

You can do this with the `setHgap()` and `setVgap()` methods in card layouts, flow layouts, border layouts, and grid layouts as well as by using the `ipadx` and `ipady` members in grid bag layouts.

Creating Your Own Layout Manager

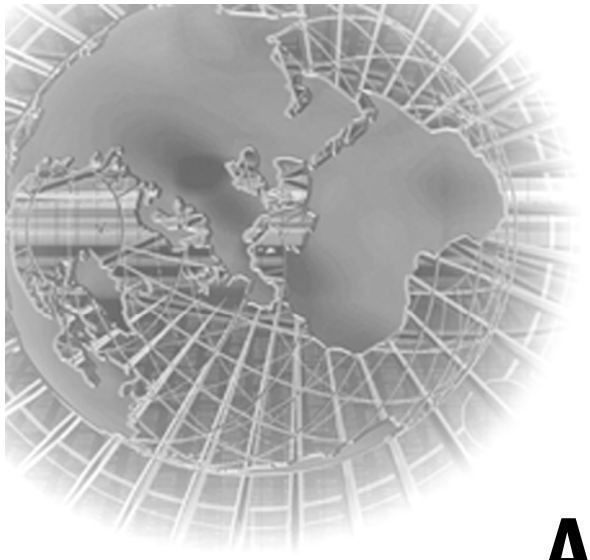
You can lay out the components in a container yourself if you uninstall any layout manager from the container (see the introduction to this chapter for details). In fact, you can even create your own layout manager if you implement the `LayoutManager2` interface in a class (the `LayoutManager` interface does not support constraints). Here are the methods you'll have to override from the `LayoutManager2 interface`:

- `public void addLayoutComponent(String name, Component component)`—It adds a component to the layout. It's called by the `Container` class. Override this method if you want to know the names of the components to layout.
- `public void removeLayoutComponent(Component component)`—It removes a component from the layout.
- `public float getLayoutAlignmentX(Container target)` and `public float getLayoutAlignmentY(Container target)`—It specifies how to align components in the x and y directions; returning a value of 0 aligns the component at the origin, returning 1 aligns it as far as possible from the origin, and returning 0.5 centers the component.
- `public void invalidateLayout(Container target)`—It invalidates a layout, which means you should discard any cached information if this method is called.

Summary

In this chapter, the AWT controls such as text fields, buttons, checkboxes, radio buttons, and layouts have been explained briefly. Then, you came to know how to implement a layout manager by understanding different types of layout. Further, we have explained the concept of inserts and padding, and finally the creation of custom layout manager.

The next chapter discusses about AWT- lists, choices, text areas, scrollbars, and scroll panes.



8

AWT—Lists, Choices, Text Areas, Scroll bars, and Scroll Panes

If you need an immediate solution to:	*See page:*
Using Text Areas	248
Replacing Text in Text Areas	251
Searching and Selecting Text in Text Areas	252
Using Lists	253
Using Multiple-Selection Lists	257
Using Choice Controls	260
Using Scroll bars	265
Scroll bars and Border Layouts	269
Using Scroll Panes	272

In Depth

This chapter is all about a number of important AWT components—lists, choices, text areas, scroll bars, and scroll panes. These components are all fundamental to AWT programming—they are familiar to nearly all GUI users—and we'll take a look at them in overview in this section. Note that one of the major themes of this chapter is *scrolling*; all the controls in this chapter support scrolling in one way or another, letting the user move around large documents with ease.

Lists

As the name suggests, a list control presents the user with a list of items—text strings—from which one can select. In windowing environments, space is often at a premium, so it is a good idea to place items in lists, because using scroll bars, you can hide long lists in short list components. The user can select the item in the list he or she wants, double-click it, and initiate some action. Lists can also support multiple selections at the same time, using the Shift and Ctrl keys. However, if you support multiple selections, you have to think about how to use the mouse in a list control. Clicking an item selects it, so the user can't just make multiple selections and then double-click one to initiate some action because when one item is double-clicked, all the others are deselected. To solve this problem, Sun suggests you to use some other event, such as a button click, to initiate an action when dealing with multiple selections.

Choices

Choice controls also present the user with a list of items to select from, but there's a difference—choice controls are even more compact than lists. They look much like text fields—although you can't edit the text in them—with a small button on the right that has a downward-pointing arrow. When the user clicks the arrow, a list opens displaying all the available choices, and the user can select from one of them. After the selection is made, the choice control closes the list and displays the current selection. If the user opens the list again, the current selection is highlighted. Note that choice controls are only designed to display one choice at a time, which means you can't select more than one item in the list at a time. Using Shift+click and Ctrl+click is the same as clicking an item. Like the other components in this chapter, choice controls support scrolling—if the list of items is long, scroll bars will appear on the side of the list automatically.

Text Areas

Text areas are much like text fields, except they support text in two-dimensions, with both rows and columns, so you can display considerably more text in them. You indicate the size in rows and columns (as measured in characters) that you want the text area to have when you create it. The result is just like a text field that has the specified number of rows and columns. You use text areas when you want to work with entire documents, not just single lines of text. As with the other components in this chapter, text areas support scrolling. You can specify whether you want a horizontal scroll bar, a vertical scroll bar, both, or none. If you don't have a horizontal scroll bar (and only if you don't have a horizontal scroll bar), the word-wrapping feature is automatically enabled.

Scroll bars

Scroll bars are, of course, the zenith of scrolling controls, and every GUI user knows about them. The user uses the mouse to manipulate the thumb—also called the *scroll box*, *elevator*, or *bubble*—in a scroll bar to select a value from a continuous range. You set that range when you create the scroll bar, and after the user selects a new value, you can catch scroll bar events to read that new value. The user can also click the arrow buttons at the ends of a scroll bar to increment or decrement the scroll bar's setting by an amount you specify. The user can also click the scroll bar's track (that is, the part that's not the thumb or the arrow buttons) to increment or decrement the scroll bar's setting by an amount you specify.

Scroll Panes

You might want to use scroll bars to scroll other components, such as a long text field, but Java controls either support scroll bars themselves or don't support scrolling at all. To make scrolling of other components easier, Java provides the `ScrollPane` class. You use scroll panes by adding components to them and letting the scroll panes handle the scrolling. After you have added a large component to a scroll pane, only part of it is visible at once, and you can use the scroll pane's scroll bars to scroll to other parts of that component.

That's it. Now that we have reviewed what's in this chapter, it is time to turn to the "Immediate Solutions" section.

Immediate Solutions

Using Text Areas

"Uh-oh," says the Novice Programmer, "I have got a problem. I have been writing my novel in a Java text field, and...." You smile and say, "And you have run out of space?" "Well," says the NP, "it is just not very convenient to write an entire novel as one line of text." "Try text areas," you suggest.

A text area is much like a two-dimensional text field; in fact, that's its big advantage —you can display whole documents in text areas, including word wrapping at the end of lines. You can also use scroll bars to move through the text. Note, however, that if you have enabled a horizontal scroll bar, word wrapping will be turned off. Here's the inheritance diagram for the AWT `TextArea` class:

```
java.lang.Object
      |____java.awt.Component
             |____java.awt.TextComponent
                    |____java.awt.TextArea
```

You'll find the fields of the `TextArea` class, which you use in the `TextArea` class's constructors, in Table 8.1, its constructors in Table 8.2, and its methods in Table 8.3.

Table 8.1: Fields of the TextArea class

Field	Does this
`static int SCROLLBARS_BOTH`	It creates and displays both vertical and horizontal scroll bars
`static int SCROLL BARS_HORIZONTAL_ONLY`	It creates and displays a horizontal scroll bar only
`static int SCROLLBARS_NONE`	It indicates not to create any scroll bars for the text area
`static int SCROLL BARS_VERTICAL_ONLY`	It creates and displays a vertical scroll bar only

Table 8.2: Constructors of the TextArea class

Constructor	Does this
`TextArea()`	It constructs a new text area with the empty string as text
`TextArea(int rows, int columns)`	It constructs a new empty text area with the given number of rows and columns and empty string as text
`TextArea(String text)`	It constructs a new text area with the given text
`TextArea(String text, int rows, int columns)`	It constructs a new text area with the given text and with the given number of rows and columns
`TextArea(String text, int rows, int columns, int scrollbars)`	It constructs a new text area with the given text, rows and columns, and scroll bar visibility

Table 8.3: Methods of the TextArea class

Method	Does this
`void addNotify()`	It creates the text area's peer
`void append(String str)`	It appends the given text to the text area's current text
`void appendText(String str)`	It is deprecated and replaced by `append(String)`
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this `TextArea`
`int getColumns()`	It gets the number of columns in the text area
`Dimension getMinimumSize()`	It gets the minimum size of the text area
`Dimension getMinimumSize(int rows, int columns)`	It gets the minimum size of the text area with given number of rows and columns

Table 8.3: Methods of the TextArea class	
Method	**Does this**
`Dimension getPreferredSize()`	It gets the preferred size of the text area
`Dimension getPreferredSize (int rows, int columns)`	It gets the preferred size of the text area with given number of rows and columns
`int getRows()`	It gets the number of rows in the text area
`int getScrollbarVisibility()`	It gets an enumerated value that indicates the scroll bars used by the text area
`void insert(String str, int pos)`	It inserts the given text at the given position in the text area
`void insertText (String str, int pos)`	It is deprecated and replaced by `insert(String, int)`
`Dimension minimumSize()`	It is deprecated and replaced by `getMinimumSize()`
`Dimension minimumSize (int rows, int columns)`	It is deprecated and replaced by `getMinimumSize(int, int)`
`protected String paramString()`	It returns a string representing the state of the text area
`Dimension preferredSize()`	It is deprecated and replaced by `getPreferredSize()`
`Dimension preferredSize(int rows, int columns)`	It is deprecated and replaced by `getPreferredSize(int, int)`
`void replaceRange (String str, int start, int end)`	It replaces text between the start and the end positions with the given text
`void replaceText (String str, int start, int end)`	It is deprecated and replaced by `replaceRange(String, int, int)`
`void setColumns(int columns)`	It sets the number of columns
`void setRows(int rows)`	It sets the number of rows

There are several ways to place text in a text area. For example, you can use the `insert()` method to insert the text at a specific character position (starting at 0) in the string that the text area holds, or you can use the `append()` method to append the text to the end of the current text. Let's look at an example. In this case, we'll create a text area with 10 rows and 20 columns (note that these dimensions are measured in characters). We'll also add two radio buttons that the user can select. Here's the code:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
        CODE=TextAreaDemo1.class
        WIDTH=200
        HEIGHT=200>
</APPLET> */
public class TextAreaDemo1 extends Applet implements ItemListener {
  TextArea area1;
  CheckboxGroup group;
  Checkbox check1,check2;
  public void init() {
        area1 = new TextArea(10,20);
        add(area1);
        group = new CheckboxGroup();
        check1 = new Checkbox("Male", false, group);
        add(check1);
        check1.addItemListener(this);
        check2 = new Checkbox("Female", false, group);
        add(check2);
        check2.addItemListener(this);
  }
  public void itemStateChanged(ItemEvent e) {
```

```
            area1.setText("You selected "+((Checkbox)e.getItemSelectable()).getLabel());
        }
    }
```

The result appears in Figure 8.1. You can see appropriate message ("You selected Male" or "You selected Female") appears in the text area on selecting either of the two radio buttons, `Male` or `Female`.

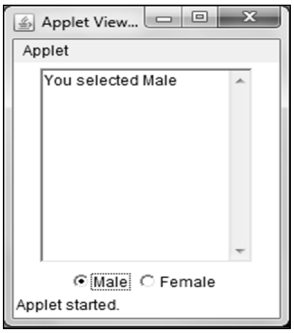


Figure 8.1: Using a text area

Let's look an example on manipulating text areas:

```
    import java.awt.*;
    import java.applet.*;
/* <APPLET
        CODE=TextAreaDemo2.class
        WIDTH=200
        HEIGHT=200>
    </APPLET> */
    public class TextAreaDemo2 extends Applet {
        public void init() {
String text = "Jack and Jill\n" + "went up the hill\n" +
"to fetch a pail of water\n" + "Jack fell down\n" +
"and broke his crown\n" + "and Jill came tumbling after";
TextArea area1 = new TextArea(text, 5, 20);
add(area1); }
}
```

The result appears in Figure 8.2. You can do even more with text areas—for example, you can select and replace text. See the next solution for details.



Figure 8.2: Manipulating text area

You can find the process of using text fields in detail in Chapter 7.

Replacing Text in Text Areas

The Product Support Specialist is unhappy, as usual, and says, "Users have a complaint about the new word processor you have written in Java." "Yes?" you ask. "They want to be able to select and replace text," the PSS says. "Some people," you say, "are never satisfied."

You can use the `replaceRange()` method to replace a range of text in a text area:

```
void replaceRange(String str, int start, int end);
```

Here, `str` is the string you want to replace the old text with, `start` is the beginning of the range of characters to replace, and `end` is the end of the range.

Here's an example in which we display the text "Kogent Learning Solutions" in a text area. The user can select any or all of that text and click a button to replace the selected text with the same converted to upper case. We start by creating the new text area and initializing it with the text "Kogent Learning Solutions":

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
         CODE=ReplaceTextDemo.class
         WIDTH=200
         HEIGHT=200 >
</APPLET> */
public class ReplaceTextDemo extends Applet implements ActionListener {
TextArea area1;
Button button1;
public void init() {
area1 = new TextArea("Kogent Learning Solutions", 10, 20);
add(area1);
button1 = new Button("Replace");
add(button1);
button1.addActionListener(this); }
```

When the user clicks the button, we can determine what text is selected by using the `getSelectionStart()` and `getSelectionEnd()` methods and then replace the selected text with same text in uppercase, like this:

```
public void actionPerformed (ActionEvent e) {
   if(e.getSource() == button1) {
   String text = area1.getText();
   int start = area1.getSelectionStart();
   int end = area1.getSelectionEnd();
   String sub = text.substring(start,end);
   area1.replaceRange(sub.toUpperCase(), start, end);}
   }
}
```

The result appears in Figure 8.3. Besides replacing text, you can also search for it—see the next solution.

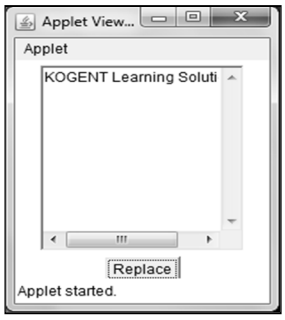


Figure 8.3: Replacing selected text in a text area

Searching and Selecting Text in Text Areas

The Product Support Specialist is back and unhappy again. "Now users want to be able to search for text when they use your word processor," the PSS reports. "What next?" you ask, exasperated. "Spell checking?"

The `TextArea` class does not support any direct methods for searching text, but the `String` class does. Therefore, you can copy the text in a text area into a `String` object and search that object for the text you want by using the `indexOf()` method.

Here's an example in which the user clicks a button to find and select the word *applet* in the text "This is an applet." in a text area. We start by creating a new text area, initializing it with the text "This is an applet."

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
        CODE=SearchAndSelect.class
        WIDTH=200
        HEIGHT=200 >
</APPLET> */
public class SearchAndSelect extends Applet implements ActionListener {
TextArea area1; Button button1;
public void init() {
        area1 = new TextArea("This is an applet.", 10, 20,
  TextArea.SCROLLBARS_BOTH);
  add(area1);
  button1 = new Button("Find \"applet\"");
  add(button1);
  button1.addActionListener(this); }
. . .
}
```

Now when the user clicks the button, the text in the text area is loaded into a string called `s`:

```
public void actionPerformed (ActionEvent e) {
if(e.getSource() == button1) {
        String s = area1.getText();
        . . . }
}
```

Now the text "applet" can be searched for using the `String` class's `indexOf()` method:

```
public void actionPerformed (ActionEvent e) {
if(e.getSource() == button1) {
        String s = area1.getText();
        String s2 = new String("applet");
        int location = s.indexOf(s2);
        . . . }
}
```

After the string "applet" is found, the `TextArea` class's `select()` method (inherited from the AWT `TextComponent` class) is used to highlight that text. Here's how you use `select()` in general:

```
void select(int selectionStart, int selectionEnd);
```

Here's how we use the `select()` method in code:

```
public void actionPerformed (ActionEvent e) {
if(e.getSource() == button1) {
String s = area1.getText();
String s2 = new String("applet");
int location = s.indexOf(s2);
area1.select(location, location + s2.length()); }
}
```

The result of this code appears in Figure 8.4:

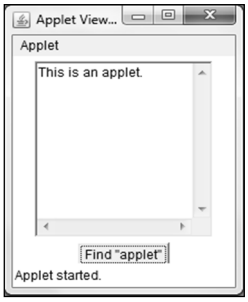


Figure 8.4: Finding and selecting text in a text area

When the user clicks the button, the code finds and highlights the text "applet" in the text area.

Using Lists

The Product Support Specialist is back and says, "There's a problem." "Yes?" you ask. "About your catalogue of all the classical music CDs in your most recent program—there's more than a hundred-thousand listings." "That's right," you say proudly. "Of course, I guess that it makes the list a little tall on the screen." "About 2,000 feet tall," says the PSS.

If you have a long list of items that you want to present, consider by using a list control. You can fill a list with many text items (using the `List` class's `add()` method) and present only a few at a time—you can choose how many—to the user. The user can highlight an item in the list by clicking the item and can initiate some action by double-clicking that item. In fact, the user can select multiple items in a list and initiate some action, typically by clicking a button. Here's the inheritance diagram for the `List` class:

```
java.lang.Object
  |____java.awt.List
        |____java.awt.Component
```

You'll find the constructors for the `List` class in Table 8.4 and the methods for this class in Table 8.5. Take a close look at the methods table; a lot of useful ones are listed, such as `getItemCount()`, which returns the number of items in the list.

Table 8.4: Constructors of the List class

Constructor	Does this
`List()`	It creates a new scrolling list
`List(int rows)`	It creates a new scrolling list with the specified number of visible lines
`List(int rows, boolean multipleMode)`	It creates a new scrolling list with the specified number of rows. Multiple selection is enabled if `multipleMode` is true

Table 8.5: Methods of the List class

Method	Does this
`void add(String item)`	It adds the given item to the end of the list
`void add(String item, int index)`	It adds the given item to the list at the position given by the `index`
`void addActionListener (ActionListener l)`	It adds the given action listener to receive action events from the list
`void addItem(String item)`	It is deprecated and replaced by `add(String)`
`void addItem (String item, int index)`	It is deprecated and replaced by `add(String, int)`
`void addItemListener (ItemListener l)`	It adds the given item listener to receive item events from the list

Table 8.5: Methods of the List class

Method	Does this
`void addNotify()`	It creates the peer for the list
`boolean allowsMultipleSelections()`	It is deprecated and replaced by `isMultipleMode()`
`void clear()`	It is deprecated and replaced by `removeAll()`
`int countItems()`	It is deprecated and replaced by `getItemCount()`
`void delItem(int position)`	It is deprecated and replaced by `remove(String)` and `remove(int)`
`void delItems(int start, int end)`	It is deprecated and no longer available for public use; is expected to be kept only as a package private method
`void deselect(int index)`	It deselects the item at the given index
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this `List`
`ActionListener[] getActionListener()`	It returns an array of all the action listeners registered on this list
`String getItem(int index)`	It gets the item associated with the given index
`int getItemCount()`	It gets the number of items in the list
`ItemListener[] getItemListeners()`	It returns an array of all the item listeners registered on this list
`String[ ] getItems()`	It gets the items in the list
`<T extends EventListener> T[] getListeners(Class<T> listenerType)`	It returns an array of all objects currently registered as *Foo*`Listener`s upon this `List`
`Dimension getMinimumSize()`	It gets the minimum size of the scrolling list
`Dimension getMinimumSize (int rows)`	It gets the minimum screen dimensions for a list with the given number of rows
`Dimension getPreferredSize()`	It gets the preferred size of this list
`Dimension getPreferredSize (int rows)`	It gets the preferred dimensions for a list with the given number of rows
`int getRows()`	It gets the number of visible lines in the list
`int getSelectedIndex()`	It gets the index of the selected item on the list
`int[] getSelectedIndexes()`	It gets the selected indexes on the list
`String getSelectedItem()`	It gets the selected item on this list
`String[] getSelectedItems()`	It gets the selected items on this list
`Object[] getSelectedObjects()`	It gets the selected items on the list in an array of objects
`int getVisibleIndex()`	It gets the index of the item that was last made visible by the method `makeVisible`
`boolean isIndexSelected (int index)`	It determines whether the given item in this list is selected
`boolean isMultipleMode()`	It determines whether the list allows multiple selections
`boolean isSelected (int index)`	It is deprecated and replaced by `isIndexSelected(int)`
`void makeVisible(int index)`	It makes the item at the given index visible at the top of the list
`Dimension minimumSize()`	It is deprecated and replaced by `getMinimumSize()`
`Dimension minimumSize(int rows)`	It is deprecated and replaced by `getMinimumSize(int)`
`protected String paramString()`	It gets the parameter string representing the state of this list
`Dimension preferredSize()`	It is deprecated and replaced by `getPreferredSize()`
`Dimension preferredSize(int rows)`	It is deprecated and replaced by `getPreferredSize(int)`
`protected void processActionEvent (ActionEvent e)`	It processes action events occurring on this list by sending them to registered `ActionListener` objects

Table 8.5: Methods of the List class

Method	Does this
`protected void processEvent(AWTEvent e)`	It processes events in this list
`protected void processItemEvent (ItemEvent e)`	It processes item events occurring on this list by sending them to registered `ItemListener` objects
`void remove(int position)`	It removes the item at the given position from the list
`void remove(String item)`	It removes the first occurrence of an item from the list
`void removeActionListener (ActionListener l)`	It removes the given action listener so that it no longer receives action events from the list
`void removeAll()`	It removes all items from the list
`void removeItemListener (ItemListener l)`	It removes the given item listener so that it no longer receives item events from the list
`void removeNotify()`	It removes the peer for the list
`void replaceItem (String newValue, int index)`	It replaces the item at the given index in the list with the new string
`void select(int index)`	It selects the item at the given index in the list
`void setMultipleMode(boolean b)`	It sets the flag that determines whether the list allows multiple selections
`void setMultipleSelections (boolean b)`	It is deprecated and replaced by `setMultipleMode(boolean)`

Let's look at an example. In this case, we'll add six color names to a list control, making only three of them visible at once. When the user double-clicks an item, background color of the applet changes accordingly. We start by creating the list control and making only three items visible, like this:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
        CODE=ListDemo.class
        WIDTH=200
        HEIGHT=200>
</APPLET> */
public class ListDemo extends Applet implements ActionListener {
        List list1;
        public void init() {
        list1 = new List(3);
  . . .
  }
```

Now we add six list items by using the `add()` method:

```
public void init() {
        list1 = new List(3);
  list1.add("Red");
  list1.add("Green");
  list1.add("Orange");
  list1.add("Pink");
  list1.add("Magenta");
  list1.add("Yellow");
  add(list1);
  . . .
}
```

After adding the list to the layout, we add a listener to it. It turns out that lists use `ActionListener` interfaces, just as buttons do. Here's how we add this to the code:

```
public void init() {
```

```
      list1 = new List(3);
            list1.add("Red");
      list1.add("Green");
      list1.add("Orange");
      list1.add("Pink");
      list1.add("Magenta");
      list1.add("Yellow");
      add(list1);
      list1.addActionListener(this);
    }
```

Now we'll need to add an `actionPerformed` method to the main applet class to handle double clicks (this method is not called for single clicks, which just select items in a list):

```
    public void actionPerformed(ActionEvent e) { . . . }
```

In this case, we'll get the item the user double-clicked by using the `List` class's `getSelectedItem` method (the first click of the double click selects the item) and change applet's background color using the Java `switch` statement, like this:

```
    public void actionPerformed(ActionEvent e) {
    if(e.getSource() == list1) {
      String color = ((List) e.getSource()).getSelectedItem();
      switch(color)  {
      case "Red" : setBackground(Color.red); break;
      case "Green" : setBackground(Color.green); break;
      case "Orange" : setBackground(Color.orange); break;
      case "Pink"    : setBackground(Color.pink); break;
      case "Magenta" : setBackground(Color.magenta); break;
      case "Yellow" : setBackground(Color.yellow);
      };
     }
    }
    }
```

The result of this code appears in Figure 8.5. Also note that only three items of the six are visible in the list because the actual list is larger than the number of visible items. The list control has automatically added a vertical scroll bar.



Figure 8.5: Using a list

You can do more with lists—for example, you can let the user make multiple selections. We'll take a look at that in the next solution.

Using Multiple-Selection Lists

"Uh-oh," says the Novice Programmer, "I have got a problem. My new program lets users Order groceries by using a list, but so far they can only order one item at a time." "No problem," you say. "Just enable multiple selection in a list."

If you pass a second parameter with the value true to the `List` class's constructor, you'll create a list component that can support multiple selection. Let's look at an example. In this case, we'll let the user select multiple items, and when he/she clicks a button, we'll list the selected items on the applet's status bar. We start by creating and populating a multiple-selection list:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
        CODE=MultipleSelection.class
        WIDTH=300
        HEIGHT=200 >
</APPLET> */
public class MultipleSelection extends Applet implements ActionListener {
List list1;
Button button1;
String selections[];
public void init() {
   list1 = new List(3, true);
   list1.add("Red");
   list1.add("Green");
   list1.add("Orange");
   list1.add("Pink");
   list1.add("Magenta");
   list1.add("Yellow");
   add(list1);
. . . }
}
```

Next, we add an `ActionListener` interface to the button and then add the button the user can click to display the selections made in the list:

```
public void init() {
   list1 = new List(3, true);
   list1.add("Red");
   list1.add("Green");
   list1.add("Orange");
   list1.add("Pink");
   list1.add("Magenta");
   list1.add("Yellow");
   add(list1);
   button1 = new Button("Show Selected");
   button1.addActionListener(this);
   add(button1);
}
```

Here, we need to use a button to show what multiple selections have been made (note that we don't let the user initiate an action involving multiple selections by using the mouse in the list because clicking or double-clicking any item will deselect all the other items).

When the user clicks the button, the `List` component's `getSelectedItems()` method is used to get a `String` array of the selected items:

```
public void actionPerformed(ActionEvent e) {
if(e.getSource() == button1) {
selections = list1.getSelectedItems();
. . . }
}
```

Next, we loop over that array, adding all the items to a string that starts "The selected items are:":

```
public void actionPerformed(ActionEvent e) {
String selected = new String("The selected items are:");
if(e.getSource() == button1) {
selections = list1.getSelectedItems();
for(int loopIndex = 0; loopIndex < selections.length;loopIndex++)
{ selected += " " + selections[loopIndex]; }
  . . . }
}
```

Finally, we place the string with all the selections on the applet's status bar:

```
public void actionPerformed(ActionEvent e) {
String selected = new String("The selected items are:");
if(e.getSource() == button1) {
selections = list1.getSelectedItems();
for(int loopIndex = 0; loopIndex < selections.length;loopIndex++)
{
  selected += " " + selections[loopIndex]; }
  showStatus(selected); }
}
```

And that's it. The result appears in Figure 8.6. As you can see in this figure, the user can make multiple selections in this new list; and when he/she clicks the button, the selections made will appear on the status bar of the applet.



Figure 8.6: Using multiple selections in a list

Here is another example having list box, as shown in Figure 8.7, and a button to remove the listener of list box:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
         CODE= ListItemDemo1.class
         WIDTH=200
         HEIGHT=200 >
</APPLET> */
public class ListItemDemo1 extends Applet implements ItemListener,
ActionListener {
TextField text1, text2;
List list1;
Button button1;
public void init() {
text1 = new TextField(20);
text2 = new TextField(20);
button1 = new Button("Remove Listener");
add(text1);
add(text2);
list1 = new List(4, true);
list1.add("First");
```

```
list1.add("Second");
list1.add("Third");
list1.add("Fourth");
list1.add("Fifth");
list1.add("Sixth");
list1.add("Seventh");
list1.add("Eighth");
list1.add("Ninth");
add(list1);
add(button1);
list1.addItemListener(this);
button1.addActionListener(this); }
public void itemStateChanged(ItemEvent e) {
if(e.getItemSelectable() == list1) {
  String sel=  ((List)e.getItemSelectable()).getSelectedItem();
if(sel!=null){
text1.setText("You selected: " +
((List)e.getItemSelectable()).getSelectedItem()); } }
}
public void actionPerformed(ActionEvent e) {
if(e.getSource() == button1) {
ItemListener[] al = list1.getItemListeners();
for (int i=0; i<al.length; i++) {
list1.removeItemListener(al[i]);
text2.setText("Listener Removed"); }
}
}
}
```

The result appears in Figure 8.7. As you can see in this figure, users can select the items from the list box; and when they do, the program displays that item in the text fields. But when the user clicks on "Remove Listener" button, listener on list will be removed and user will not be able to select the item from the list.



Figure 8.7: Removing listeners

Preceding example can be used with `ActionListener` interface as:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.ListSelectionEvent;
/* <APPLET
  CODE=ListItemDemo2.class
  WIDTH=200
  HEIGHT=250 >
```

```
</APPLET> */
public class ListItemDemo2 extends Applet implements ActionListener {
  TextArea text1;
  TextField text2;
  List list1;
  Button button1;
  public void init() {
        text1 = new TextArea(5,20);
        text2 = new TextField(20);
        button1 = new Button("Remove Listener");
        add(text1);
        add(text2);
        list1 = new List(4, true);
        list1.add("First");
        list1.add("Second");
        list1.add("Third");
        list1.add("Fourth");
        list1.add("Fifth");
        list1.add("Sixth");
        list1.add("Seventh");
        list1.add("Eighth");
        list1.add("Ninth");
        add(list1);
        add(button1);
        list1.addActionListener(this);
        button1.addActionListener(this); }
  public void valueChanged(ListSelectionEvent e) {
        if(e.getValueIsAdjusting())
        return;
        text1.setText("");
        Object[] items = list1.getSelectedItems();
        for(int i=0; i<items.length; i++)
        text1.append(items[i] +"\n"); }
  public void actionPerformed(ActionEvent e) {
        if(e.getSource() == button1) {
        ActionListener[] al = list1.getActionListeners();
        for (int i=0; i<al.length; i++) {
                list1.removeActionListener(al[i]);
                text2.setText("Listener Removed"); }
        }
  }
}
```

Using Choice Controls

"Good Lord!" says the Novice Programmer, "The Big Boss is cutting costs again, and he says all the controls must be half their original size. But the list control I'm using is already too small!" "No problem," you say. "Just use a choice control instead."

Choice controls are much like list controls that just display one item. To display the entire list in a choice control, you click the button that appears on the right side of the control. This causes a drop-down list to appear, and you can select an item with the mouse. Then, the list closes. The current choice appears in the choice control when the list is closed (note that you can't make multiple selections in a choice control). Here's the inheritance diagram for choice controls:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Choice
```

You'll find the constructor for the `Choice` class in Table 8.6 and the methods of this class in Table 8.7.

Table 8.6: The constructor of the Choice class

Constructor	Does this
`Choice()`	It creates a new choice control

Table 8.7: Methods of the Choice class

Method	Does this
`void add(String item)`	It adds an item to this choice control
`void addItem(String item)`	It is obsolete as of Java 2 version 1.1
`void addItemListener (ItemListener l)`	It adds the given item listener to receive events from this choice control
`void addNotify()`	It creates the choice control's peer
`int countItems()`	It is deprecated and replaced by `getItemCount()`
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this `Choice`
`String getItem(int index)`	It gets the string at the given index in this choice control
`int getItemCount()`	It gets the number of items in this choice control
`ItemListener[] getItemListeners()`	It returns an array of all the item listeners registered on this choice control
`<T extends EventListener> T[] getListeners(Class<T> listenerType)`	It returns an array of all the objects currently registered as *Foo*`Listener`s upon this choice control
`int getSelectedIndex()`	It gets the index of the currently selected item
`String getSelectedItem()`	It gets a string representation of the current choice
`Object[] getSelectedObjects()`	It gets an array (of length 1) containing the currently selected item
`void insert(String item, int index)`	It inserts the item into this choice control at the given position
`protected String paramString()`	It gets the parameter string representing the state of this choice control
`protected void processEvent(AWTEvent e)`	It processes events on this choice control
`protected void processItemEvent (ItemEvent e)`	It processes item events occurring on this choice control by sending them to any registered `ItemListener` objects
`void remove(int position)`	It removes an item from the choice control at the given position
`void remove(String item)`	It removes the first occurrence of an item from the choice control
`void removeAll()`	It removes all items from the choice control
`void removeItemListener (ItemListener l)`	It removes an item listener so that it no longer gets item events from this choice control
`void select(int pos)`	It sets the selected item in this choice control to the given position
`void select(String str)`	It sets the selected item in this choice control to the item with the name that's the same as the given string

Let's look at an example. In this case, we'll add a choice control to a program and use the control's `add()` method to add items to its internal list. When the user makes a selection, we'll display the new selection in a text field.

We start by creating a new choice control and populating its internal list with choices (note that you don't have a lot of options when creating choice controls—there's only one constructor, and it doesn't take any parameters).

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
```

```
/* <APPLET
        CODE=ChoiceDemo1.class
        WIDTH=200
        HEIGHT=200 >
</APPLET> */
public class ChoiceDemo1 extends Applet implements ItemListener, ActionListener {
TextField text1, text2;
Choice choice1;
Button button1;
public void init() {
text1 = new TextField(20);
text2 = new TextField(20);
button1 = new Button("Remove Listener");
add(text1);
add(text2);
choice1 = new Choice();
choice1.add("First");
choice1.add("Second");
choice1.add("Third");
choice1.add("Fourth");
choice1.add("Fifth");
choice1.add("Sixth");
choice1.add("Seventh");
choice1.add("Eighth");
choice1.add("Ninth");
choice1.add("Tenth");
choice1.add("Eleventh");
choice1.add("Twelfth");
. . .
}
```

With a large number of items like this, the choice control will add a vertical scroll bar to the list automatically.

Now we'll handle events in the choice control. You use `ItemListener` interfaces with choice controls and `ActionListener` interfaces, as you do with list controls. The `ItemListener` interface has only one method, `itemStateChanged()`:

```
void itemStateChanged(ItemEvent e)
```

This method is passed as an object of the `ItemEvent` class. This class's fields appear in Table 8.8, its constructor in Table 8.9, and its methods in Table 8.10.

Table 8.8: Fields of the ItemEvent class

Field	Does this
`static int DESELECTED`	It indicates that a selected item was unselected
`static int ITEM_FIRST`	It indicates the first number in the range of IDs used for item events
`static int ITEM_LAST`	It indicates the last number in the range of IDs used for item events
`static int ITEM_STATE_CHANGED`	It indicates that an item's state changed
`static int SELECTED`	It indicates that an item was selected

Table 8.9: The constructor of the ItemEvent class

Constructor	Does this
`ItemEvent(ItemSelectable source, int id, Object item, int stateChange)`	It constructs an `ItemEvent` object

Table 8.10: Methods of the ItemEvent class	
Method	**Does this**
`Object getItem()`	It gets the item affected by the event
`ItemSelectable etItemSelectable()`	It gets the originator of the event
`int getStateChange()`	It gets the type of state change (that is, selected or deselected)
`String paramString()`	It gets a string identifying this item event

We next add an `ItemListener` interface to the choice control, like this:

```
    public void init() {
    text1 = new TextField(20);
    text2 = new TextField(20);
    button1 = new Button("Remove Listener");
    add(text1);
    add(text2);
    choice1 = new Choice();
    choice1.add("First");
    choice1.add("Second");
    choice1.add("Third");
    choice1.add("Fourth");
    choice1.add("Fifth");
    choice1.add("Sixth");
    choice1.add("Seventh");
    choice1.add("Eighth");
    choice1.add("Ninth");
    choice1.add("Tenth");
    choice1.add("Eleventh");
    choice1.add("Twelfth");
    add(choice1);
    add(button1);
    choice1.addItemListener(this);
    button1.addActionListener(this);
    }
```

When the user does make a selection from the choice control's internal list, we'll use the choice control's `getSelectedItem()` method to determine which item was selected and then display it in a text field:

```
    public void itemStateChanged(ItemEvent e) {
    if(e.getItemSelectable() == choice1) {
    text1.setText("You selected: " +
    ((Choice)e.getItemSelectable()).getSelectedItem()); }
    }
    public void actionPerformed(ActionEvent e) {
    if(e.getSource() == button1) {
    ItemListener[] al = choice1.getItemListeners();
    for (int i=0; i<al.length; i++) {
    choice1.removeItemListener(al[i]);
    text2.setText("Listener Removed"); }
    }
    }
    }
```

The result appears in Figure 8.8. As you can see in the figure, users can select an item from the choice control's list; and when they do, the program displays that item in the text field.

But on clicking the button "Remove Listener", listener on the choice list will be removed and, as a result, user will not be able to select the items from choice's list.



Figure 8.8: Using a choice control

Look at this example which creates two Choice menus. One selects the color of shirt. The other selects the color of trousers:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/* <APPLET
        CODE=ChoiceDemo2.class
        WIDTH=200
        HEIGHT=200 >
</APPLET> */
public class ChoiceDemo2 extends Applet implements ItemListener {
   Choice shirt, trouser;
   String msg = "";
   public void init() {
                shirt = new Choice();
        trouser = new Choice();
   shirt.add("Red Shirt");
   shirt.add("Blue Shirt");
   shirt.add("Cream Shirt");
   shirt.add("Black Shirt");
        trouser.add("Gray Trouser");
        trouser.add("Blue Trouser");
        trouser.add("Black Trouser");
        add(shirt);
        add(trouser);
        shirt.addItemListener(this);
        trouser.addItemListener(this); }
   public void itemStateChanged(ItemEvent ie) { repaint(); }
// Display current selections.
public void paint(Graphics g) {
msg = "Shirt: ";
   msg += shirt.getSelectedItem();
   g.drawString(msg, 6, 120);
   msg = "Trouser: ";
   msg += trouser.getSelectedItem();
   g.drawString(msg, 6, 140);  }
}
```

The output of the `ChoiceDemo2` class has been given in Figure 8.9:



Figure 8.9: Using Choice menus

Using Scroll bars

The Novice Programmer appears and says, "I have a problem. I want to let users select the drawing color in my new program, but they are getting tired of typing in color values in hexadecimal format." You smile and say, "Why not use scroll bars instead? They are the perfect component when you want to let users select from a continuous numeric range." "Sounds perfect," the NP says.

Every GUI user is familiar with scroll bars. In Java, scroll bars consist of arrows (the buttons at each end of the scroll bar), a thumb or bubble (the scrollable box you slide), and a track (the part of the scroll bar you slide the thumb in). The AWT supports scroll bars with the `Scrollbar class`, which has the following inheritance diagram:

```
java.lang.Object
   |____java.awt.Component
         |____java.awt.Scrollbar
```

You'll find the constructors for the `Scrollbar` class in Table 8.11 and its methods in Table 8.12.

Table 8.11: Constructors of the Scrollbar class

Constructor	Does this
`Scrollbar()`	It creates a new vertical scroll bar
`Scrollbar(int orientation)`	It creates a new scroll bar with the given orientation
`Scrollbar(int orientation, int value int visible, int minimum, int maximum)`	It creates a new scroll bar with the given orientation, initial value, page size, and minimum and maximum values

Table 8.12: Methods of the Scrollbar class

Method	Does this
`void addAdjustmentListener (AdjustmentListener l)`	It adds the given adjustment listener to receive adjustment events
`void addNotify()`	It creates the scroll bar's peer
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this `Scrollbar`
`AdjustmentListener[] getAdjustmentListeners()`	It returns an array of all adjustment listeners registered on this scroll bar
`int getBlockIncrement()`	It gets the block increment of the scroll bar

Table 8.12: Methods of the Scrollbar class

Method	Does this
`int getLineIncrement()`	It is deprecated and replaced by `getUnitIncrement()`
<T extends EventListener[]> T[] getListeners `(Class<T> listenerType)`	It returns an array of all the objects currently registered as *Foo*`Listeners` upon this scroll bar
`int getMaximum()`	It gets the maximum value of the scroll bar
`int getMinimum()`	It gets the minimum value of the scroll bar
`int getOrientation()`	It gets the orientation of the scroll bar
`int getPageIncrement()`	It is deprecated and replaced by `getBlockIncrement()`
`int getUnitIncrement()`	It gets the unit increment for this scroll bar
`int getValue()`	It gets the current value of the scroll bar
`boolean getValueIsAdjusting()`	It returns True if the value is in the process of changing as a result of action being taken by the user
`int getVisible()`	It is deprecated and replaced by `getVisibleAmount()`
`int getVisibleAmount()`	It gets the visible amount of the scroll bar
`protected String paramString()`	It gets a string representing the state of the scroll bar
`protected void processAdjustmentEvent (AdjustmentEvent e)`	It processes adjustment events occurring in this scroll bar by sending them to any registered `AdjustmentListener` objects
`protected void processEvent(AWTEvent e)`	It processes events on the scroll bar
`void removeAdjustmentListener (AdjustmentListener l)`	It removes the given adjustment listener so that it no longer receives adjustment events from the scroll bar
`void setBlockIncrement (int v)`	It sets the block increment for the scroll bar
`void setLineIncrement (int v)`	It is deprecated and replaced by `setUnitIncrement(int)`
`void setMaximum (int newMaximum)`	It sets the maximum value
`void setMinimum (int newMinimum)`	It sets the minimum value
`void setOrientation (int orientation)`	It sets the orientation
`void setPageIncrement (int v)`	It is deprecated and replaced by `setBlockIncrement()`
`void setUnitIncrement (int v)`	It sets the unit increment
`void setValue(int newValue)`	It sets the value of the scroll bar
`void setValueIsAdjusting (boolean b)`	It sets the `valueIsAdjusting` property
`void setValues(int value, int visible, int minimum, int maximum)`	It sets values for the four properties of the scroll bar: `value`, `visibleAmount`, `minimum`, and `maximum`
`void setVisibleAmount (int newAmount)`	It sets the visible amount of the scroll bar

Scroll bars don't use `ActionListener` or `ItemListener` interfaces; instead, they use `AdjustmentListener` interfaces. The `AdjustmentListener` interface has only one method, `adjustmentValueChanged()`:

```
void adjustmentValueChanged(AdjustmentEvent e)
```

This method, `adjustmentValueChanged()`, passes an object of class `AdjustmentEvent`. You'll find the fields of the `AdjustmentEvent` class in Table 8.13, its constructor in Table 8.14, and its methods in Table 8.15.

Note, in particular, that you can use the `AdjustmentEvent` class's `getAdjustmentType()` method to determine what kind of scroll bar event occurred, as specified by the fields you see in Table 8.13.

Table 8.13: Fields of the AdjustmentEvent class	
Field	**Does this**
static int ADJUSTMENT_FIRST	It indicates the first integer ID for the range of adjustment event IDs
static int ADJUSTMENT_LAST	It indicates the last integer ID for the range of adjustment event IDs
static int ADJUSTMENT_VALUE_CHANGED	It indicates the adjustment value changed event
static int BLOCK_DECREMENT	It indicates the block decrement adjustment type
static int BLOCK_INCREMENT	It indicates the block increment adjustment type
static int TRACK	It indicates the absolute tracking adjustment type
static int UNIT_DECREMENT	It indicates the unit decrement adjustment type
static int UNIT_INCREMENT	It indicates the unit increment adjustment type

Table 8.14: The constructor of the AdjustmentEvent class	
Constructor	**Does this**
`AdjustmentEvent(Adjustable source, int id, int type, int value)`	It constructs an `AdjustmentEvent` object with the given `Adjustable` source, event type, adjustment type, and value
`AdjustmentEvent(Adjustable` source, int id, int type, int value, boolean isAdjusting)	It constructs an AdjustmentEvent object with the specified `Adjustable` source, event type, adjustment type, and value

Table 8.15: Methods of the AdjustmentEvent class	
Method	**Does this**
`Adjustable getAdjustable()`	It gets the `Adjustable` object where this event originated
`int getAdjustmentType()`	It gets the type of adjustment that caused the value changed event
`int getValue()`	It gets the current value in the adjustment event
`boolean getValueIsAdjusting()`	It returns True if this is one of the multiple adjustment events
`String paramString()`	It returns a string representing the state of this `Event`

Let's look at an example. Here, we'll just add two scroll bars to a program and change the applet's background color when the user adjusts the scroll bars. When you construct a scroll bar, you can specify its orientation (horizontal or vertical), its initial value, a page size, and its numeric range. The *page size* indicates how big you want the thumb to be (it is customary to let the size of the thumb give an indication of the total range—you use a smaller thumb for a longer range and a larger thumb for a shorter range, as you see in word processors when they work with documents of varying sizes). Here's how we add a scroll bar to a program, giving it a horizontal orientation, an initial value of 1, a page size of 20, and a range from 1 to 150:

```
import java.applet.Applet;
import java.awt.event.*;
import java.awt.*;
/* <APPLET
        CODE=ScrollbarDemo.class
        WIDTH=400
        HEIGHT=100 >
</APPLET> */
public class ScrollbarDemo extends Applet implements AdjustmentListener {
Scrollbar scroll1, scroll2;
```

Note that we also add an `AdjustmentListener` interface to this new scroll bar.

TIP

You might think that this scroll bar can return any value in the range of 1 to 150; but in fact, the thumb must also be represented in the track, so the scroll bar can only return values from 1 to 150 minus the page size (20, in this case), which equals 130. That's something to think about when you are creating scroll bars.

You can also configure the scroll bar by using the `setUnitIncrement()` and `setBlockIncrement()` methods. The `setUnitIncrement()` method sets the amount the scroll bar's setting changes when the user clicks an arrow button (the default is 1), and the `setBlockIncrement()` method sets the amount the setting changes when the user clicks the scroll bar track (the default is 10). The `init()` method looks like this:

```
public void init() {
scroll1 = new Scrollbar(Scrollbar.HORIZONTAL, 1, 20, 1, 150);
add(scroll1);
scroll1.addAdjustmentListener(this);
scroll2 = new Scrollbar(Scrollbar.VERTICAL, 1, 20, 1, 150);
add(scroll2);
scroll2.addAdjustmentListener(this);
}
```

All that's left is to update the background color when the user uses the scroll bars. We do that in the `adjustmentValueChanged()` method, checking the `AdjustmentEvent` object's `getAdjustable()` method to make sure we are dealing with one of the scroll bars and using that scroll bar's `getValue()` method to display the new settings in the text field. Here's how this looks in code:

```
public void adjustmentValueChanged(AdjustmentEvent e) {
if(e.getAdjustable() == scroll1 ||e.getAdjustable() == scroll2) {

  int value1 = scroll1.getValue();
  int value2 = scroll2.getValue();

  setBackground(new Color(100,value1,value2));
  showStatus("Horizontal: "+value1+" Vertical: "+value2);
}
}
}
```

The result appears in Figure 8.10.

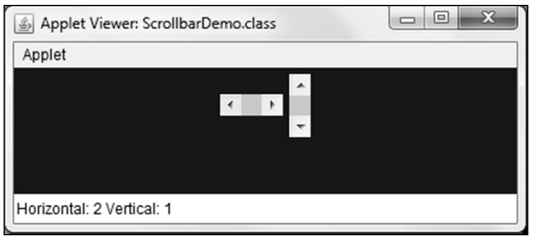


Figure 8.10: Using scroll bars

As you can see in this figure, the user can move the scroll bars, and the new settings of those scroll bars will change the background color.

That's fine if you just want to use freestanding scroll bars to let the user easily specify numbers from a continuous range, but what if you want to use scroll bars to scroll something? Next, we'll take a look at another example that scrolls text strings around an applet.

In this case, we'll move the strings "Kogent" and "Learning" around the applet as the user scrolls the scroll bar. We start by creating the scroll bar and painting the strings at the location (`x`, 80) and (40, `y`) where we'll adjust both `x and y` when the scroll bar is adjusted:

```
import java.applet.Applet;
import java.awt.event.*;
import java.awt.*;
```

```
/* <APPLET
        CODE=ScrollText.class
        WIDTH=200
        HEIGHT=200 >
</APPLET> */
public class ScrollText extends Applet implements AdjustmentListener {
Scrollbar scroll1;
int x = 40;
int y = 40;
public void init() {
scroll1 = new Scrollbar(Scrollbar.HORIZONTAL, 1, 20, 20, 100);
add(scroll1);
scroll1.addAdjustmentListener(this); }
public void paint(Graphics g) {
g.drawString("Kogent", x, 80);
g.drawString("Learning",40, y);
}
     . . .
}
```

When scroll events occur, we'll be notified in the `adjustmentValueChanged()` method, and we can set the value in the variable `x` and `y` to reflect the new locations of the strings and redraw the applet (note that we are using the `Applet` class's `getSize()` method here to determine the width of the applet and scale the motions of the text strings accordingly):

```
public void adjustmentValueChanged(AdjustmentEvent e) {
if(e.getAdjustable() == scroll1) {
x = (int) (getSize().width * (float) scroll1.getValue() / 100);
y = (int) (getSize().width * (float) scroll1.getValue() / 100);
repaint(); }
}
```

The result appears in Figure 8.11. As you can see in this figure, the user can scroll the text strings around the applet just by manipulating the horizontal scroll bar. We'll take a closer look at scrolling text with scroll bars in the next solution.



Figure 8.11: Scrolling text strings

Scroll bars and Border Layouts

"Look," says the Novice Programmer, "now the Big Boss wants me to add horizontal and vertical scroll bars to my program, but they just don't want to stay where I put them!" "That's because you should use a border layout," you say.

Because border layouts let you add controls around the perimeter of a program, they are natural to use with scroll bars. Here's an example in which we add four scroll bars around a central panel. This panel displays the image, "`image.jpg`" at the location (`x, y`), like this:

```
class DisplayPanel extends Panel {
public int x = 0, y = 0;
```

```
    public void paint (Graphics g) {
    Image img = Toolkit.getDefaultToolkit().getImage("image.jpg");
    g.drawImage(img, x, y, this); }
    }
```

Next, we can add an object of this new panel class to the center of a border layout, surrounding it with scroll bars, like this:

```
    import java.applet.Applet;
    import java.awt.*;
    import java.awt.event.*;
    /* <APPLET
            CODE=ScrollBorderDemo1.class
            WIDTH=250
            HEIGHT=200 >
    </APPLET> */
    public class ScrollBorderDemo1 extends Applet implements AdjustmentListener {
    Scrollbar horizontal1, horizontal2, vertical1, vertical2;
    DisplayPanel display;
    public void init()
      {
            setLayout(new BorderLayout());
      horizontal1 = new Scrollbar(Scrollbar.HORIZONTAL, 1, 1, 1, 100);
      add("North", horizontal1);
      horizontal1.addAdjustmentListener(this);
      vertical1 = new Scrollbar(Scrollbar.VERTICAL, 1, 1, 1, 100);
      add("West", vertical1);
      vertical1.addAdjustmentListener(this);
      horizontal2 = new Scrollbar(Scrollbar.HORIZONTAL, 1, 1, 1, 100);
      add("South", horizontal2);
      horizontal2.addAdjustmentListener(this);
      vertical2 = new Scrollbar(Scrollbar.VERTICAL, 1, 1, 1, 100);
      add("East", vertical2);
      vertical2.addAdjustmentListener(this);
      display = new DisplayPanel();
      add("Center", display); }
      . . .
    }
```

The only trick here is that when the user moves one scroll bar, we have to move the corresponding one on the other side of the central panel to keep the scroll bars coordinated. Here's how this looks in the `adjustmentValueChanged()` method:

```
    public void adjustmentValueChanged(AdjustmentEvent e) {
    if(e.getAdjustable() == horizontal1) {
      horizontal2.setValue(horizontal1.getValue()); }
    if(e.getAdjustable() == vertical1){
      vertical2.setValue(vertical1.getValue()); }
    if(e.getAdjustable() == horizontal2) {
      horizontal1.setValue(horizontal2.getValue()); }
    if(e.getAdjustable() == vertical2) {
      vertical1.setValue(vertical2.getValue()); }
    . . .
    }
```

All that's left is to get the new scroll bar settings, adjust the `x` and `y` locations of the text string in the panel to match, and then redraw the panel. The process looks like this:

```
    public void adjustmentValueChanged(AdjustmentEvent e) {
    if(e.getAdjustable() == horizontal1) {
      horizontal2.setValue(horizontal1.getValue()); }
    if(e.getAdjustable() == vertical1){
      vertical2.setValue(vertical1.getValue()); }
```

```
    if(e.getAdjustable() == horizontal2) {
      horizontal1.setValue(horizontal2.getValue()); }
    if(e.getAdjustable() == vertical2) {
      vertical1.setValue(vertical2.getValue()); }
    display.x = (int) (getSize().width * (float) horizontal1.getValue() / 200);
    display.y = (int) (getSize().height * (float) vertical1.getValue() /150);
    display.repaint();
    }
```

And that's all it takes. The result appears in Figure 8.12. As you can see in this figure, the scroll bars appear around the perimeter of the applet.



Figure 8.12: Scroll bars in a border layout

Take a look at this code which adds four buttons and a text area to border layout:

```
    import java.awt.*;
    import java.applet.*;
    /* <applet code=ScrollBorderDemo2.class width=400 height=200>
    </applet> */
    public class ScrollBorderDemo2 extends Applet {
    public void init() {
    setLayout(new BorderLayout());
    add(new Button("Top part"), BorderLayout.NORTH);
    add(new Button("Bottom part"), BorderLayout.SOUTH);
    add(new Button("Right-hand side"), BorderLayout.EAST);
    add(new Button("Left-hand side"), BorderLayout.WEST);
    String text = "The secret of health " +
             "for both mind and body \n"+
             "is not to mourn for the \n"+
             "past, not to worry about \n"+
             "the future, not to anticipate \n"+
             "troubles, but to live the \n"+
             "present moment wisely and \n"+
             "earnestly. \n\n"+
             "[Siddhartha Gautama, 563 - 483 BC] \n\n";
    add(new TextArea(text), BorderLayout.CENTER); }
    }
```

The result appears in Figure 8.13. There's another way to make scrolling text in panels easy—you can use scroll panes, which is coming up next.

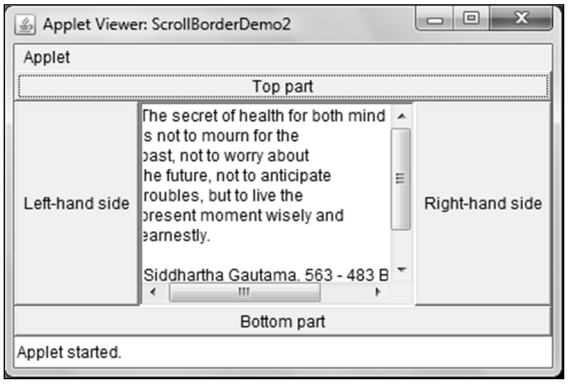


Figure 8.13: Border layout and Scroll bars

Using Scroll Panes

"I want to scroll some text," the Novice Programmer says, "but setting up the scroll bars is too much work. Isn't there an easier way?" "There sure is," you say, "you can use scroll panes." "Great!" says the NP.

You can add a component to a scroll pane, and the scroll pane will let you scroll that component around. If the component is larger than the scroll pane, only part of the component will be visible at any one time. Here's the inheritance diagram for the `ScrollPane` class:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____java.awt.ScrollPane
```

You'll find the fields of the `ScrollPane` class in Table 8.16, its constructors in Table 8.17, and its methods in Table 8.18.

Table 8.16: Fields of the ScrollPane class

Field	Does this
`static int SCROLL BARS_ALWAYS`	It indicates that horizontal/vertical scroll bars should always be shown
`static int SCROLL BARS_AS_NEEDED`	It indicates that horizontal/vertical scroll bars should be shown only when the size of the child is greater than the size of the scroll pane
`static int SCROLL BARS_NEVER`	It indicates that horizontal/vertical scroll bars should never be shown

Table 8.17: Constructors of the ScrollPane class

Constructor	Does this
`ScrollPane()`	It creates a new scroll pane
`ScrollPane(int scrollbarDisplayPolicy)`	It creates a new scroll pane by using a display policy

Table 8.18: Methods of the ScrollPane class

Method	Does this
`protected void addImpl (Component comp, Object constraints, int index)`	It adds the given component to this scroll pane
`void addNotify()`	It creates the scroll pane's peer
`void doLayout()`	It lays out this container by resizing the contained component to its preferred size

Table 8.18: Methods of the ScrollPane class

Method	Does this
`protected boolean eventTypeEnabled(int type)`	If wheel scrolling is enabled, it returns True for `MouseWheelEvents`
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this `ScrollPane`
`Adjustable getHAdjustable()`	It gets the `ScrollPaneAdjustable` object that represents the state of the horizontal scroll bar
`int getHScrollbarHeight()`	It gets the height that would be occupied by a horizontal scroll bar (independent of whether it is currently displayed or not in the scroll pane)
`int getScrollbarDisplayPolicy()`	It gets the display policy for the scroll bars
`Point getScrollPosition()`	It gets the current `x,y` position within the child, which is displayed at the origin of the scrolled panel's view port
`Adjustable getVAdjustable()`	It gets the `ScrollPaneAdjustable` object that represents the state of the vertical scroll bar
`Dimension getViewportSize()`	It gets the current size of the scroll pane's view port
`int getVScrollbarWidth()`	It gets the width that would be occupied by a vertical scroll bar
`boolean isWheelScrollingEnabled()`	It indicates whether or not scrolling will take place in response to mouse wheel
`void layout()`	It is deprecated and replaced by `doLayout()`
`String paramString()`	It returns a string representing the state of this scroll pane
`void printComponents (Graphics g)`	It prints the component in this scroll pane
`protected void processMouseWheelEvent (MouseWheelEvent e)`	It processes mouse wheel events that are delivered to this by scrolling a proper amount
`void setLayout (LayoutManager mgr)`	It sets the layout manager for this container
`void setScrollPosition (int x, int y)`	It scrolls to the given position within the child component
`void setScrollPosition(Point p)`	It scrolls to the given position within the child component
`void setWheelScrollingEnabled (boolean handlewheel)`	It enables or disables scrolling in response to movement of the mouse wheel

Let's look at an example. It is easy enough to just add a component to a scroll pane; you simply use the `ScrollPane` class's `add()` method. Here's how this looks when we add a text area to a scroll pane:

```
import java.applet.Applet;
import java.awt.*;
/* <APPLET
        CODE=ScrollPaneDemo1.class
        WIDTH=350
        HEIGHT=200>
</APPLET> */
public class ScrollPaneDemo1 extends Applet {
ScrollPane scrollpane1;
TextArea area1;
public void init() {
scrollpane1 = new ScrollPane();
area1 = new TextArea(10,40);
scrollpane1.add(area1);
setLayout(new BorderLayout());
add(scrollpane1); }
}
```

The result of `ScrollPaneDemo1.java` appears in Figure 8.14. As you can see in this figure, the user ends up scrolling text around inside a scroll pane.

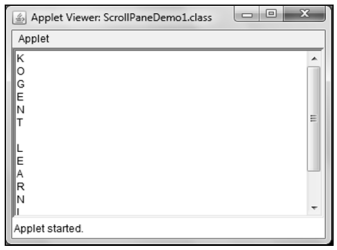


Figure 8.14: Scrolling the text inside a scroll pane

There is another example in which we add 25 labels to a panel and pass the panel reference in the `ScrollPane`'s `add()` method. Both the horizontal and vertical scroll bars automatically appear inside the applet. Here's the code:

```
import java.applet.Applet;
import java.awt.*;
/*<APPLET
        CODE=ScrollPaneDemo2.class
        WIDTH=200
        HEIGHT=120>
</APPLET> */
public class ScrollPaneDemo2 extends Applet {
ScrollPane scrollpane1;
Label labels[];
Panel panel1;
public void init() {
        scrollpane1 = new ScrollPane(ScrollPane.SCROLLBARS_ALWAYS);
        panel1 = new Panel();
        panel1.setLayout(new GridLayout(5,5));
        labels = new Label[25];
        for(int i=0;i<25;i++) {
         labels[i] = new Label("Kogent"); panel1.add(labels[i]); }
        setLayout(new BorderLayout());
        scrollpane1.add(panel1);
        add(scrollpane1);
  }
}
```

Figure 8.15 shows the result of `ScrollPaneDemo2.java`.

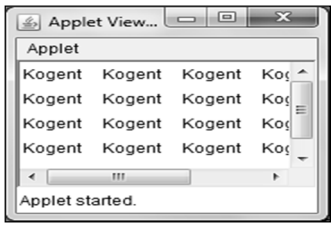


Figure 8.15: Scrolling labels inside a scroll pane

Summary

In this chapter, we provided you with a brief introduction on lists, choices, text areas, scrollbars, and scroll panes. Then, you learned how to use the text areas, replace text in text areas, and search text in the text areas. You also came to know about the use of lists, multiple selection lists, followed by the use of choice controls. Toward the end, you studied about the scrollbars and scroll panes.

In the next chapter, we will discuss about AWT- graphics, images, text, and fonts.



9

AWT—Graphics, Images, Text, and Fonts

If you need an immediate solution to:	*See page:*
Using the Mouse	277
Using the Keyboard	279
Using Fonts	283
Using Images	289
Resizing Images	291
Drawing Graphics	292
Selecting Colors	300
Using Canvases	303
Using the ImageObserver Interface	305
Using the MediaTracker Class	306
Working Pixel by Pixel: The PixelGrabber and MemoryImageSource Classes	308
Brightening Images	311
Converting Images to Grayscale	312
Embossing Images	313

In Depth

This chapter is all about some very powerful topics in Java—graphics, images, text handling, and working with fonts. Java is a very visual language, and all these areas are popular with programmers. We'll start with an overview of these items.

Graphics

The graphics capability in AWT is pretty solid, and it is based on the huge `Graphics` class. You can use this class to draw all kinds of figures—lines, points, rectangles, ovals, polygons, and more. You can also select colors and drawing modes as well as fill figures in with color. You'll see all this in this chapter, including a special component—the `Canvas` component—that exists expressly, so you can draw in it.

Images

Images are the objects of `Image` class found in the `java.awt` package. Various classes have been provided in the `java.awt.image` package to implement images. You'll see various different capabilities of AWT in this chapter, such as loading images in distinct image formats, resizing images, waiting until images get loaded completely, drawing images offscreen before displaying (known as *double buffering*), and animating images.

In fact, you can even gain access to the individual pixels in images—copying images, brightening them, converting them to grayscale, and giving them an engraved appearance.



You'll learn all about animating images and double buffering when we discuss multithreading in Chapter 19.

Text and Fonts

You might be surprised to see text and fonts in a chapter on graphics, but when you draw text directly, not inserting it into a control, such as a text field, you are creating a graphic, and Java treats it as such. In this chapter, you'll see that there's quite a bit to working with text as graphics. For example, you'll see how to set the font and font style—such as italics or bold—of text as well as how to measure the screen length of a string of text so you can center it in a window.

The Keyboard and Mouse

In this chapter, we'll be writing code that can display text directly, without any controls, such as text fields and text areas, which means we'll have to read that text directly from the keyboard. Therefore, we'll take a look at handling keyboard input here. In addition, when you are letting the user create graphics, the mouse is a very handy tool to have—in fact, we'll create a mouse-driven drawing program in this chapter—so we'll also take a look at working with the mouse in code. In fact, we'll start this chapter by taking a look at using the mouse and keyboard. Since there's a lot more coming up here, it's time to turn to the "Immediate Solutions" section.

Immediate Solutions

Using the Mouse

"OK," says the Novice Programmer, "in my program, the user can select text with Ctrl+Alt+F8, move the insertion point with Shift+Alt+F3, and. . ." "Wait a minute," you say. "Have you ever thought about adding mouse support to your program? It might make things easier for the user." "Hmm," says the NP thoughtfully. You can work with the mouse using two AWT interfaces—`MouseListener`, which handles mouse clicks, presses, and releases as well as instances when the mouse enters a component and then leaves it, and `MouseMotionListener`, which handles mouse movements and drag operations. You can find the methods of the `MouseListener` interface in Table 9.1 and the methods of the `MouseMotionListener` interface in Table 9.2.

TIP

Bear in mind that you need to override all the methods in an interface if you want to implement that interface, which can get a little tedious in the case of MouseListener, which has five methods. To make things easier, you can use adapter classes such as MouseAdapter instead and just override the methods you want to use. For more on adapter classes, see Chapter 6.

Table 9.1: Methods of the MouseListener interface

Method	Does this
`void mouseClicked (MouseEvent e)`	It is called when the mouse has been clicked (pressed and released) on a component
`void mouseEntered (MouseEvent e)`	It is called when the mouse enters a component
`void mouseExited (MouseEvent e)`	It is called when the mouse exits a component
`void mousePressed (MouseEvent e)`	It is called when a mouse button has been pressed on a component
`void mouseReleased (MouseEvent e)`	It is called when a mouse button has been released on a component

Table 9.2: Methods of the MouseMotionListener interface

Method	Does this
`void mouseDragged (MouseEvent e)`	It is called when a mouse button is pressed on a component and then dragged
`void mouseMoved (MouseEvent e)`	It is called when the mouse button has been moved over a component (with no buttons down)

Each of the mouse interface methods is passed an object of class `MouseEvent`, and the inheritance diagram for that class looks like this:

```
java.lang.Object
  |____java.util.EventObject
        |____java.awt.AWTEvent
              |____java.awt.event.ComponentEvent
                    |____java.awt.event.InputEvent
                          |____java.awt.event.MouseEvent
```

You can find the constructors of the `MouseEvent` class in Table 9.3.

Table 9.3: Constructors of the MouseEvent class

Constructor	Does this
`MouseEvent(Component source, int id, long when, int modifiers, int x, int y, int clickCount, boolean popupTrigger )`	It constructs a `MouseEvent` object with the indicated source component, type, modifiers, coordinates, click count, and popupTrigger flag
`MouseEvent(Component source, int id, long when, int modifiers, int x, int y, int clickCount, boolean popupTrigger, int button)`	It constructs a `MouseEvent` object with the indicated source component, type, time, modifiers, coordinates, click count, popupTrigger flag, and button number

Table 9.3: Constructors of the MouseEvent class	
Constructor	**Does this**
`MouseEvent(Component source, int id, long when, int modifiers, int x, int y, int xAbs, int yAbs, int clickCount, boolean popupTrigger, int button)`	It constructs a `MouseEvent` object with the indicated source component, type, time, modifiers, coordinates, absolute coordinates, click count, popupTrigger flag, and button number

The `MouseEvent` class adds only fields to its base classes, and you can find the fields of this class in Table 9.4.

Table 9.4: Fields of the MouseEvent class	
Field	**Does this**
`static int BUTTON1`	It indicates mouse button#1; used by `getButton()`
`static int BUTTON2`	It indicates mouse button#2; used by `getButton()`
`static int BUTTON3`	It indicates mouse button#3; used by `getButton()`
`static int MOUSE_CLICKED`	It indicates the "mouse clicked" event
`static int MOUSE_DRAGGED`	It indicates the "mouse dragged" event
`static int MOUSE_ENTERED`	It indicates the "mouse entered" event
`static int MOUSE_EXITED`	It indicates the "mouse exited" event
`static int MOUSE_FIRST`	It indicates the first number in the range of IDs used for mouse events
`static int MOUSE_LAST`	It indicates the last number in the range of IDs used for mouse events
`static int MOUSE_MOVED`	It indicates the "mouse moved" event
`static int MOUSE_PRESSED`	It indicates the "mouse pressed" event
`static int MOUSE_RELEASED`	It indicates the "mouse released" event
`static int MOUSE_WHEEL`	It indicates the mouse wheel event
`static int NOBUTTON`	It indicates no mouse buttons; used by `getButton()`

Let's look at an example. This applet, called `MousePlay.java`, will display most of what the mouse can do. To catch particular mouse actions, you just override the corresponding mouse listener method. To get the current location of the mouse from a `MouseEvent` object, you can use the `getX()` and `getY()` methods. To determine which button was pressed, you can use the `MouseEvent` class's `getModifiers()` method and then `And` the result with these fields from the `InputEvent` class: `ALT_GRAPH_MASK`, `ALT_MASK`, `BUTTON1_MASK`, `BUTTON2_MASK`, `BUTTON3_MASK`, `CTRL_MASK`, `META_MASK`, and `SHIFT_MASK`. Here's what this applet looks like, with all this work put together:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
   CODE=MousePlay.class
   WIDTH=300
   HEIGHT=200 >
</APPLET> */
class ShapePanel extends Panel implements MouseListener,MouseMotionListener{
   MousePlay play;
   ShapePanel(MousePlay mp){
         play=mp;
   }
   public void mousePressed(MouseEvent e){
         play.showStatus("Mouse Pressed");
   }
   public void mouseClicked(MouseEvent e) {
   if((e.getModifiers() & InputEvent.BUTTON1_MASK) ==InputEvent.BUTTON1_MASK) {
   play.showStatus("Left Mouse Button Clicked at "+e.getX()+","+e.getY());
```

```
      }else{play.showStatus("Right Mouse Button Clicked at "+e.getX()+","+e.getY());    }
          }
      public void mouseReleased(MouseEvent e) {
            play.showStatus("Mouse Released at "+e.getX()+","+e.getY());}
      public void mouseEntered(MouseEvent e) {
            play.showStatus("Mouse Entered");}
      public void mouseExited(MouseEvent e) {
            play.showStatus("Mouse Exited");}
      public void mouseDragged(MouseEvent e) {
            play.showStatus("Mouse Dragged");}
      public void mouseMoved(MouseEvent e) {
            play.showStatus("Mouse Moved");}
   }
   public class MousePlay extends Applet
   {
      Panel panel;
      public void init() {
            panel= new Panel();
            panel.setBackground(Color.pink);
            panel.setPreferredSize(new Dimension(160,160));
            add(panel);
            panel.addMouseListener(new ShapePanel(this));
            panel.addMouseMotionListener(new ShapePanel(this)); }
   }
```

You can see this applet at work in Figure 9.1. The frame contains a pink-colored panel with `MouseListener` and `MouseMotionListener` added to it. Appropriate message is displayed in the applet's status bar corresponding to different mouse events. In this way, the applet lets you know what's going on.

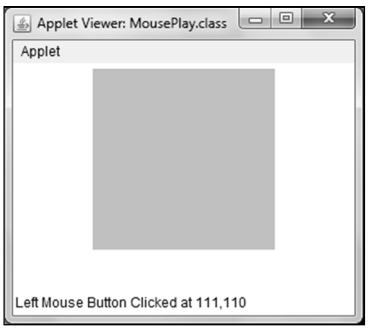


Figure 9.1: Using the mouse

Using the Keyboard

"Say," says the Novice Programmer, "I'm writing a word processor, *SuperDuper TextPro*, in Java and I want to read text directly from the keyboard. How does that work?" "With the key listener," you say. "I knew it'd be something like that," the NP says.

You use the `KeyListener` interface to work with the keyboard, and you can find the methods of the `KeyListener` interface in Table 9.5.

Table 9.5: Methods of the KeyListener interface

Method	Does this
`void keyPressed(KeyEvent e)`	It is called when a key is pressed
`void keyReleased (KeyEvent e)`	It is called when a key is released
`void keyTyped(KeyEvent e)`	It is called when a key is typed

Note that there are three different key events—`KeyPressed`, `KeyReleased`, and `KeyTyped`. You usually use the `KeyTyped` event when working with the keyboard, because you can use the `getKeyChar()` method in

KeyTyped to get the Unicode character that was typed. In the keyPressed() and keyReleased() methods, on the other hand, you can use the getKeyCode() (not getKeyChar()) method to get a *virtual key code*; this code just tells you what key was pressed or released— you are responsible for determining whether the Shift, Ctrl, or other key was pressed down, which you can do with the KeyEvent object passed to the key event methods. To determine which modifier keys (such as Shift) were pressed down, you can use the KeyEvent object's getModifiers() method and then And the result with the following fields from the InputEvent class: ALT_GRAPH_MASK, ALT_MASK, CTRL_MASK, META_MASK, and SHIFT_MASK.

In general, it's not very easy to work with virtual key codes because there's a separate constant you have to use for each key as returned by getKeyCode()—for example, VK_F1 for the F1 key, VK_A for the character A, VK_5 for the number 5, and so on, as enumerated in the fields of the KeyEvent class, which you can see in Table 9.6. You can also find the constructors for the KeyEvent class in Table 9.7 and its methods in Table 9.8. Here's the inheritance diagram for the KeyEvent class:

```
java.lang.Object
  |____java.util.EventObject
        |____java.awt.AWTEvent
              |____java.awt.event.ComponentEvent
                    |____java.awt.event.InputEvent
                          |____java.awt.event.KeyEvent
```

Table 9.6: Fields of the KeyEvent class

static char CHAR_UNDEFINED	static int KEY_FIRST	static int KEY_LAST
static int EY_LOCATION_LEFT	static int KEY_LOCATION_NUMPAD	static int KEY_LOCATION_RIGHT
static int KEY_LOCATION_STANDARD	static int KEY_STANDARD_UNKNOWN	static int KEY_PRESSED
static int KEY_RELEASED	static int KEY_TYPED	static int VK_0 to VK_9
static int VK_A to VK_Z	static int VK_ACCEPT	static int VK_ADD
static int VK_AGAIN	static int VK_ALL_CANDIDATES	static int VK_ALPHANUMERIC
static int VK_ALT	static int VK_ALT_GRAPH	static int VK_AMPERSAND
static int VK_ASTERISK	static int VK_AT	static int VK_BACK_QUOTE
static int VK_BACK_SLASH	static int VK_BACK_SPACE	static int VK_BEGIN
static int VK_BRACELEFT	static int VK_BRACERIGHT	static int VK_CANCEL
static int VK_CAPS_LOCK	static int VK_CIRCUMFLEX	static int VK_CLEAR
static int VK_CLOSE_BRACKET	static int VK_CODE_INPUT	static int VK_COLON
static int VK_COMMA	static int VK_COMPOSE	static int VK_CONTEXT_MENU
static int VK_CONTROL	static int VK_CONVERT	static int VK_COPY
static int VK_CUT	static int VK_DEAD_ABOVEDOT	static int VK_DEAD_ABOVERING
static int VK_DEAD_ACUTE	static int VK_DEAD_BREVE	static int VK_DEAD_CARON
static int VK_DEAD_CEDILLA	static int VK_DEAD_CIRCUMFLEX	static int VK_DEAD_DIAERESIS
static int VK_DEAD_DOUBLEACUTE	static int VK_DEAD_GRAVE	static int VK_DEAD_IOTA
static int VK_DEAD_MACRON	static int VK_DEAD_OGONEK	static int VK_DEAD_SEMIVOICED_SOUND
static int VK_DEAD_TILDE	static int VK_DEAD_VOICED_SOUND	static int VK_DECIMAL
static int VK_DELETE	static int VK_DIVIDE	static int VK_DOLLAR
static int VK_DOWN	static int VK_END	static int VK_ENTER
static int VK_EQUALS	static int VK_ESCAPE	static int VK_EURO_SIGN
static int VK_EXCLAMATION_MARK	static int VK_F1 to VK_F24	static int VK_FINAL

Table 9.6: Fields of the KeyEvent class		
static int VK_FIND	static int VK_FULL_WIDTH	static int VK_GREATER
static int VK_HALF_WIDTH	static int VK_HELP	static int VK_HIRAGANA
static int VK_HOME	static int VK_INPUT_METHOD_ON_OFF	static int VK_INSERT
static int VK_INVERTED _ EXCLAMATION_MARK	static int VK_JAPANESE_ HIRAGANA	static int VK_JAPANESE_KATAKANA
static int VK_JAPANESE_ROMAN	static int VK_KANA	static int VK_KANA_LOCK
static int VK_KANJI	static int VK_KATAKANA	static int VK_KP_DOWN
static int VK_KP_LEFT	static int VK_KP_RIGHT	static int VK_KP_UP
static int VK_LEFT	static int VK_LEFT_PARENTHESIS	static int VK_LESS
static int VK_META	static int VK_MINUS	static int VK_MODECHANGE
static int VK_MULTIPLY	static int VK_NONCONVERT	static int VK_NUM_LOCK
static int VK_NUMBER_SIGN	static int VK_NUMPAD0 to VK_NUMPAD9	static int VK_OPEN_BRACKET
static int VK_PAGE_DOWN	static int VK_PAGE_UP	static int VK_PASTE
static int VK_PAUSE	static int VK_PERIOD	static int VK_PLUS
static int VK_PREVIOUS_CANDIDATE	static int VK_PRINTSCREEN	static int VK_PROPS
static int VK_QUOTE	static int VK_QUOTEDBL	static int VK_RIGHT
static int VK_RIGHT_PARENTHESIS	static int VK_ROMAN_CHARACTERS	static int VK_SCROLL_LOCK
static int VK_SEMICOLON	static int VK_SEPARATER	static int VK_SEPARATOR
static int VK_SHIFT	static int VK_SLASH	static int VK_SPACE
static int VK_STOP	static int VK_SUBTRACT	static int VK_TAB
static int VK_UNDEFINED	static int VK_UNDERSCORE	static int VK_UNDO
static int VK_UP	static int VK_WINDOWS	

Table 9.7: Constructors of the KeyEvent class	
Constructor	**Does this**
KeyEvent(Component source, int id, long when, int modifiers, int keyCode)	It is deprecated
KeyEvent(Component source, int id, long when, int modifiers, int keyCode, char keyChar)	It creates a KeyEvent object
KeyEvent(Component source, int id, long when, int modifiers, int keyCode, char keyChar, int keyLocation)	It creates a KeyEvent object

Table 9.8: Methods of the KeyEvent class	
Method	**Does this**
int getExtendedKeyCode()	It gets an extended key code for the event
static int getExtendedKeyCodeForChar (int c)	It gets an extended key code for a Unicode character
char getKeyChar()	It gets the character associated with the key in this event
int getKeyCode()	It gets the integer keycode associated with the key in this event
int getKeyLocation()	It gets the location of thw key that originated this event
static String getKeyModifiersText (int modifiers)	It gets a string describing the modifier key(s), such as "Shift" or "Ctrl+Shift"

Table 9.8: Methods of the KeyEvent class	
Method	**Does this**
`static String getKeyText (int keyCode)`	It gets a string describing the keycode, such as "HOME", "F1", or "A"
`boolean isActionKey()`	It determines whether the key in this event is an "action" key
`String paramString()`	It returns a parameter string identifying this event
`void setKeyChar (char keyChar)`	It sets the `keyChar` value to indicate a character
`void setKeyCode (int keyCode)`	It sets the `keyCode` value to indicate a physical key
`void setModifiers (int modifiers)`	It is deprecated

Let's look at an example. This simple applet contains two text fields: one in which the user enters the text and the second simultaneously displays the entered text. Here, the `KeyEvent` is trapped, and the applet uses the `getKeyChar()` method to actually read the keys typed. Every time the user types a key, the corresponding character is added to the end of the displayed string. Here's what the applet looks like:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
/* <APPLET
   CODE=KeyboardPlay.class
   WIDTH=300
   HEIGHT=200>
</APPLET> */
public class KeyboardPlay extends Applet implements KeyListener {
TextField text1,text2;
public void init() {
text1 = new TextField(20);
add(text1);
text2 = new TextField(20);
add(text2);
text1.addKeyListener(this);
}
public void keyTyped(KeyEvent e) {
String text = text2.getText();
text = text + e.getKeyChar();
text2.setText(text); }
public void keyPressed(KeyEvent e) { }
public void keyReleased(KeyEvent e) { }
}
```

The result of this code appears in Figure 9.2. As you can see in the figure, the user can type text in one text field and it gets displayed in the other.

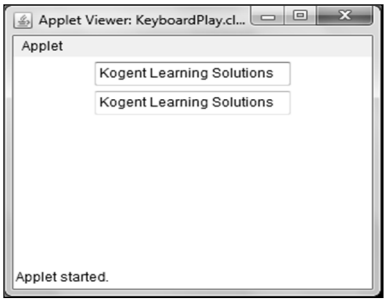


Figure 9.2: Using the keyboard

That's just the beginning of working with text. Now let's take a look at the next solution, where we start working with fonts.

You can find the process of Using Text Fields in detail in Chapter 7.

Using Fonts

"The banner you created for the Company Pride Picnic was good," the Big Boss says, "but it didn't seem to be bursting with pride." "Why not?" you ask. "Well for one thing," the BB says, "it was only a quarter of an inch tall." "Hmm," you say, "guess I better use a bigger font."

The `Font` class allows you to select a font style, such as Arial, Courier, or Roman, set font size, and specify if it's bold, italic, and so on. You'll find the fields of the `Font` class in Table 9.9, its constructors in Table 9.10, and its methods in Table 9.11.

Table 9.9: Fields of the Font class

Field	Does this
`static int BOLD`	The bold style
`static int CENTER_BASELINE`	The baseline used in ideographic scripts (such as Japanese)
`static String DIALOG`	A String constant for the canonical family name of the logical font "Dialog"
`static String DIALOG_INPUT`	A String constant for the canonical family name of the logical font "DialogInput"
`static int HANGING_BASELINE`	The baseline used when laying text using scripts, such as Devanagari
`static int ITALIC`	The italicized style
`static int LAYOUT_LEFT_TO_RIGHT`	A flag to layoutGlyphVector indicating that the text is left to right as determined by the Bidi analysis
`static int LAYOUT_NO_LIMIT_CONTEXT`	A flag to layoutGlyphVector indicating that the text in the char array after the indicated limit should not be examined
`static int LAYOUT_NO_START_CONTEXT`	A flag to layoutGlyphVector indicating that the text in the char array before the indicated limit should not be examined
`static int LAYOUT_RIGHT_TO_LEFT`	A flag to layoutGlyphVector indicating that the text is right to left as determined by the Bidi analysis
`static int MONOSPACED`	A String constant for the canonical family name of logical font "Monospaced"
`protected String name`	The logical name of this `Font`
`static int PLAIN`	The plain style
`protected float pointSize`	The point size of this `Font` in `float`
`static int ROMAN_BASELINE`	The baseline used in most roman scripts when laying out text
`static String SANS_SERIF`	A String constant for the canonical family name of logical font "SansSerif"
`static String SERIF`	A String constant for the canonical family name of logical font "Serif"
`protected int size`	The point size of this `Font`, rounded to integer
`protected int style`	The style of this `Font`
`static int TRUETYPE_FONT`	Identify a font resource of type TRUETYPE
`static int TYPE1_FONT`	Identify a font resource of type TYPE1

Table 9.10: Constructors of the Font class

Constructor	Does this
`protected Font (Font font)`	It creates a new `Font` from the indicated `font`
`Font(Map<? extends AttributedCharacterIterator .Attribute,?> attributes)`	It creates a new `Font` with the indicated attributes
`Font(String name, int style, int size)`	It creates a new `Font` from the indicated name, style, and point size

Table 9.11: Methods of the Font class

Method	Does this
`boolean canDisplay(char c)`	It checks whether the font has a glyph for the given character
`boolean canDisplay(int codepoint)`	It checks whether the font has a glyph for the given character
`int canDisplayUpTo(char[] text, int start, int limit)`	It indicates whether this font can display the characters in the given `text` starting at `start` and ending at `limit`
`int canDisplayUpTo (CharacterIterator iter, int start, int limit)`	It indicates whether this font can display the text specified by `iter` starting at `start` and ending at `limit`
`int canDisplayUpTo (String str)`	It indicates whether this font can display the given string
`static Font createFont (int fontFormat, File fontFile)`	It returns a new font using the given font type and file
`static Font createFont (int fontFormat, InputStream fontStream)`	It returns a new font with the given font type and input data
`GlyphVector createGlyphVector (FontRenderContext frc, char[] chars)`	It creates a `GlyphVector` object by mapping characters to glyphs one-to-one based on the Unicode cmap in this font
`GlyphVector createGlyphVector (FontRenderContext frc, CharacterIterator ci)`	It creates a `GlyphVector` object by mapping the given characters to glyphs one-to-one based on the Unicode cmap in this font
`GlyphVector createGlyphVector (FontRenderContext frc, int[] glyphCodes)`	It creates a `GlyphVector` object by mapping characters to glyphs one-to-one based on the Unicode cmap in this font
`GlyphVector createGlyphVector (FontRenderContext frc, String str)`	It creates a `GlyphVector` object by mapping characters to glyphs one-to-one based on the Unicode cmap in this font
`static Font decode (String str)`	It gets the font that the string `str` describes
`Font deriveFont (AffineTransform trans)`	It creates a new `Font` object by duplicating the current `Font` object and applying a new transform
`Font deriveFont(float size)`	It creates a new `Font` object by duplicating the current `Font` object and applying a new size
`Font deriveFont(int style)`	It creates a new `Font` object by duplicating the current `Font` object and applying a new style
`Font deriveFont(int style, AffineTransform trans)`	It creates a new `Font` object by duplicating the current `Font` object and applying a new style and transform
`Font deriveFont(int style, float size)`	It creates a new `Font` object by duplicating the current `Font` object and applying a new style and size
`Font deriveFont(Map attributes)`	It creates a new `Font` object by duplicating the current `Font` object and applying a new set of font attributes
`Font deriveFont(Map<? extends AttributedCharacterIterator.Attribute, ?> attributes)`	It creates a new `Font` object by duplicating the current `Font` object and applying a new set of font attributes
`boolean equals(Object obj)`	It compares this `Font` object to the given `Object`
`Map<TextAttribute,?> getAttributes()`	It gets a map of font attributes in this font
`AttributedCharacterIterator.Attribute[] getAvailableAttributes()`	It gets the keys of all the font attributes
`byte getBaselineFor(char c)`	It gets the baseline for displaying this character

Table 9.11: Methods of the Font class

Method	Does this
`String getFamily()`	It gets the family name of this font
`String getFamily(Locale l)`	It gets the family name of this font, localized for the given locale
`static Font getFont(Map<? extends AttributedCharacterIterator.Attribute,?> attributes)`	It gets a `Font` object appropriate to the attributes
`static Font getFont(String nm)`	It gets a `Font` object from the system properties list
`static Font getFont(String nm, Font font)`	It gets the given font from the system properties list
`String getFontName()`	It gets the font face name
`String getFontName(Locale l)`	It gets the font face name, localized for the given locale
`float getItalicAngle()`	It gets the italic angle of this font
`LineMetrics getLineMetrics(char[] chars, int beginIndex, int limit, FontRenderContext frc)`	It gets a `LineMetrics` object created using the given arguments
`LineMetrics getLineMetrics (CharacterIterator ci, int beginIndex, int limit, FontRenderContext frc)`	It gets a `LineMetrics` object created using the given arguments
`LineMetrics getLineMetrics(String str, FontRenderContext frc)`	It gets a `LineMetrics` object created using the given string and `FontRenderContext`
`LineMetrics getLineMetrics(String str, int beginIndex, int limit, FontRenderContext frc)`	It gets a `LineMetrics` object created using the given arguments
`Rectangle2D getMaxCharBounds (FontRenderContext frc)`	It gets the bounds for the character with the maximum bounds, as defined in the given `FontRenderContext`
`int getMissingGlyphCode()`	It gets the glyphCode used when this font does not have a glyph for a given Unicode code point
`String getName()`	It gets the font's logical name
`int getNumGlyphs()`	It gets the number of glyphs in this font
`java.awt.peer.FontPeer getPeer()`	It is deprecated. Font rendering is now supposed to be platform-independent
`String getPSName()`	It gets the postscript name of the font
`int getSize()`	It gets the point size of the font, rounded to an integer
`float getSize2D()`	It gets the point size of this font in a float value
`Rectangle2D getStringBounds(char[] chars, int beginIndex, int limit, FontRenderContext frc)`	It gets the logical bounds of the given array of characters in the given `FontRenderContext`
`Rectangle2D getStringBounds (CharacterIterator ci, int beginIndex, int limit, FontRenderContext frc)`	It gets the logical bounds of the characters indexed in the given `CharacterIterator` in the given `FontRenderContext`
`Rectangle2D getStringBounds(String str, FontRenderContext frc)`	It gets the logical bounds of the given string in the given `FontRenderContext`
`Rectangle2D getStringBounds(String str, int beginIndex, int limit, FontRenderContext frc)`	It gets the logical bounds of the given string in the given `FontRenderContext`
`int getStyle()`	It gets the style of this font
`AffineTransform getTransform()`	It gets a copy of the transform associated with this font

Table 9.11: Methods of the Font class	
Method	**Does this**
`int hashCode()`	It gets a hashcode for this font
`boolean hasLayoutAttributes()`	It checks whether the font contains attributes requiring extra processing
`boolean hasUniformLineMetrics()`	It checks whether this font has uniform line metrics
`boolean isBold()`	It indicates whether this `Font` object's style is bold
`boolean isItalic()`	It indicates whether this `Font` object's style is italic
`boolean isPlain()`	It indicates whether this `Font` object's style is plain
`boolean isTransformed()`	It indicates whether this `Font` object has a transform that affects its size in addition to the Size attribute
`GlyphVector layoutGlyphVector (FontRenderContext frc, char[] text, int start, int limit, int flags)`	It returns a new `GlyphVector`, performing a complete layout of the text, if possible
`String toString()`	It converts the `Font` object into a string representation

As if all the methods in Table 9.11 weren't enough, there're two another important classes —the `LayoutPath` class and `FontMetrics` class. The `LayoutPath` class's constructor appears in Table 9.12, its methods in Table 9.13. Here's the inheritance diagram for the `LayoutPath` class:

```
java.lang.Object
  |____java.awt.font.LayoutPath
```

The `LayoutPath` class extends `Object`. It is used for getting a mapping between locations that are relative to the baseline and points in the space of user. Along the baseline, an advance of location is coneprised, and an offset is perpendicular to the baseline at the advance. Locations are represented as a `Point2D`, where `x` is the advance and `y` is the offset.

Table 9.12: The Constructors of the LayoutPath class	
Constructor	**Does this**
`LayoutPath()`	It constructs a new `LayoutPath`

Table 9.13: Methods of the LayoutPath class	
Method	**Does this**
`abstract void pathToPoint(Point2D location, boolean preceding, Point2D point)`	It converts a location relative to the path to a point in user space
`abstract boolean pathToPoint(Point2D point, Point2D location)`	It converts a point in user coordinates to a location relative to the path

The `FontMetrics` class tells you about the physical dimensions of fonts. The `FontMetrics` class's field appears in Table 9.14, its constructor in Table 9.15, and its methods in Table 9.16. Typically, you use the `FontMetrics` class to determine the height of text while displaying multiline text. It's worthwhile going through the tables in this solution to see what's available when working with fonts.

Table 9.14: The field of the FontMetrics class	
Field	**Does this**
`protected Font font`	It determines the font from which the font metrics are created

Table 9.15: The constructor of the FontMetrics class	
Constructor	**Does this**
`protected FontMetrics (Font font)`	It creates a new `FontMetrics` object.

Table 9.16: Methods of the FontMetrics class

Method	Does this
`int bytesWidth(byte[] data, int off, int len)`	It gets the total advance width for showing the given array of bytes in this font
`int charsWidth(char[] data, int off, int len)`	It gets the total advance width for showing the given array of characters in this font
`int charWidth(char ch)`	It gets the advance width of the given character in this font
`int charWidth(int codePoint)`	It gets the advance width of the given character in this font
`int getAscent()`	It indicates the font ascent of the font described by this `FontMetrics` object
`int getDescent()`	It indicates the font descent of the font described by this `FontMetrics` object
`Font getFont()`	It gets the font described by this `FontMetrics` object
`int getHeight()`	It gets the standard height of a line of text in this font
`int getLeading()`	It indicates the standard leading of the font described by this `FontMetrics` object
`LineMetrics getLineMetrics (char[] chars, int beginIndex, int limit, Graphics context)`	It gets the `LineMetrics` object for the given character array and `Graphics` context
`LineMetrics getLineMetrics (CharacterIterator ci, int beginIndex, int limit, Graphics context)`	It gets the `LineMetrics` object for the given `CharacterIterator` and `Graphics` context
`LineMetrics getLineMetrics (String str, Graphics context)`	It gets the `LineMetrics` object for the given string and `Graphics` context
`LineMetrics getLineMetrics (String str, int beginIndex, int limit, Graphics context)`	It gets the `LineMetrics` object for the given string and `Graphics` context
`int getMaxAdvance()`	It gets the maximum advance width of any character in this font
`int getMaxAscent()`	It indicates the maximum ascent of the font described by this `FontMetrics` object
`Rectangle2D getMaxCharBounds (Graphics context)`	It gets the bounds for the character with the maximum bounds in the given `Graphics` context
`int getMaxDecent()`	It is deprecated and replaced by `getMaxDescent()`
`int getMaxDescent()`	It indicates the maximum descent of the font described by this `FontMetrics` object
`Rectangle2D getStringBounds(char[] chars, int beginIndex, int limit, Graphics context)`	It gets the bounds of the given array of characters in the given `Graphics` context
`Rectangle2D getStringBounds (CharacterIterator ci, int beginIndex, int limit, Graphics context)`	It gets the bounds of the characters indexed in the given `CharacterIterator` and `Graphics` context
`Rectangle2D getStringBounds(String str, Graphics context)`	It gets the bounds of the given string in the given `Graphics` context
`Rectangle2D getStringBounds(String str, int beginIndex, int limit, Graphics context)`	It gets the bounds of the given string in the given `Graphics` context
`int[] getWidths()`	It gets the advance widths of the first 256 characters in the font

Table 9.16: Methods of the FontMetrics class	
Method	**Does this**
`boolean hasUniformLineMetrics()`	It checks whether the font has uniform line metrics
`int stringWidth(String str)`	It gets the total advance width for showing the given string in this font
`String toString()`	It gets this `FontMetrics` object's values as a string

Let's look at an example. In this case, we'll let the user type characters and display them in `Courier` font, centered in the applet, by determining the screen size of the text using the `FontMetrics` class's `stringWidth()` and `getHeight()` methods, and the width and height of the applet with the applet's `getSize()` method. We'll also let the user specify the size of the text, as well as whether it should be in italics or bold, and set a `Font` object accordingly. To actually install the font so that when you print text it appears in that font, you use a `Graphics` object's `setFont()` method. Here's what the applet looks like:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
/* <APPLET
   CODE=FontDemo.class
   WIDTH=600
   HEIGHT=200>
</APPLET> */
public class FontDemo extends Applet implements ActionListener, KeyListener {
String text = "";
Button boldbutton, italicbutton, largebutton;
boolean bold = false;
boolean italic = false;
boolean large = false;
public void init() {
boldbutton = new Button("Bold");
italicbutton = new Button("Italic");
largebutton = new Button("Large");
boldbutton.addActionListener(this);
italicbutton.addActionListener(this);
largebutton.addActionListener(this);
add(boldbutton);
add(italicbutton);
add(largebutton);
addKeyListener(this);
requestFocus(); }
public void actionPerformed(ActionEvent event) {
if(event.getSource() == boldbutton) bold = !bold;
if(event.getSource() == italicbutton) italic = !italic;
if(event.getSource() == largebutton) large = !large;
requestFocus();
repaint(); }
public void paint(Graphics g) {
String fontname = "Courier";
int type = Font.PLAIN;
int size = 36;
Font font;
FontMetrics  metrics;
if(bold) type = type | Font.BOLD;
if(italic) type = type | Font.ITALIC;
if(large) size = 72;
font = new Font(fontname, type, size);
g.setFont(font);
metrics = getFontMetrics(font);
int xloc = (getSize().width - metrics.stringWidth(text)) / 2;
int yloc = (getSize().height + metrics.getHeight()) / 2;
```

```
g.drawString(text, xloc, yloc); }
public void keyTyped(KeyEvent e) {
text = text + e.getKeyChar();
repaint(); }
        public void keyPressed(KeyEvent e) { }
        public void keyReleased(KeyEvent e) { }
}
```

When either all the buttons such as Bold, Italic, and Large are clicked or one at a time, then the entered text appears centered in the applet with the corresponding attributes, shown in Figure 9.3. Now it's time to move on to working with images—take a look at the next solution.

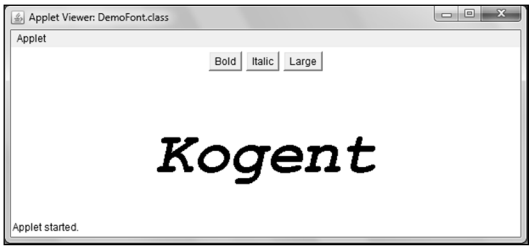


Figure 9.3: Using fonts

Using Images

The Big Boss says, "About this photo essay you wrote for the company newspaper...." "Yes?" you ask. "Where are the photos?" the BB asks. "Hmm," you say, "this looks like a job for the `Image` class." You can support images in AWT with the `Image` class, which is derived directly from `java.lang.Object`:

```
java.lang.Object
   |____java.awt.Image
```

Images are manipulated using the classes found in the `java.awt.image` package. You'll find the fields of the `Image` class in Table 9.17, its constructor in Table 9.18, and its methods in Table 9.19.

Table 9.17: Fields of the Image class

Field	Does this
`protected float accelerationPriority`	The priority for accelerating this image
`static int SCALE_AREA_AVERAGING`	Use the Area Averaging image-scaling algorithm
`static int SCALE_DEFAULT`	Use the default image-scaling algorithm
`static int SCALE_FAST`	Choose an image-scaling algorithm that gives higher priority to scaling speed than smoothness
`static int SCALE_REPLICATE`	Use the image-scaling algorithm in the `ReplicateScaleFilter` class
`static int SCALE_SMOOTH`	Choose an image-scaling algorithm that gives higher priority to scaling speed than smoothness
`static Object UndefinedProperty`	The `UndefinedProperty` object should be returned whenever a property that was not defined for a particular image is requested

Table 9.18: The constructor of the Image class

Constructor	Does this
`Image()`	It creates an `Image` object

Table 9.19: Methods of the Image class

Method	Does this
`void flush()`	It flushes all reconstructable resources used by the image

Table 9.19: Methods of the Image class	
Method	**Does this**
`float getAccelerationPriority()`	It returns the current value of the acceleration priority hint
`ImageCapabilities getCapabilities (GraphicsConfiguration gc)`	It returns an `ImageCapabilities` object that can be inquired as to the capabilities of the image on given `GraphicsConfiguration`
`abstract Graphics getGraphics()`	It creates a graphics context for drawing an offscreen image
`abstract int getHeight (ImageObserver observer)`	It indicates the height of the image
`abstract Object getProperty (String name, ImageObserver observer)`	It gets a property of this image by name
`Image getScaledInstance(int width, int height, int Hints)`	It creates a scaled version of this image
`abstract ImageProducer getSource()`	It gets the object that produces the pixels for the image
`abstract int getWidth (ImageObserver observer)`	It indicates the width of the image
`void setAccelerationPriority float priority)`	It sets a hint for this image about the importance of acceleration

To create an image object, you can use the `createImage()` method like this:

```
Image createImage(int width, int height)
```

Suppose you want to create an instance of `Img` and then call the `createImage()` method to actually make an `Image` object. This would look like this:

```
Img m = new Img();
Image test = m.createImage(200, 100);
```

At this point, the image is blank. To load an image into an applet, you can use the `Applet` class's `getImage()` method:

```
Image getImage(URL url)
Image getImage(URL url, String name)
```

Here, you can specify the URL of the image file you want to read using the `URL` class. You can create a `URL` object using the `URL` class's constructor, like this:

```
URL("http://www.oracle.com")
```

You'll learn more about this in the coming chapters. As far as this chapter is concerned, however, we'll use the `Applet` class's `getCodeBase()` and `getDocumentBase()` methods to get the URL for the applet itself and use that as the same location to find the image file.

Here's a short example that just reads in an image, `Java8.jpg`, and displays it. To read in the image, we use the `getImage()` method. To draw the image, we use the `Graphics` class's `drawImage()` method. For a lot more on the `Graphics` class, see the solution "Drawing Graphics," later in this chapter. Here's the form of `drawImage()` we'll use here, which lets you specify the image object to draw and position it in the applet:

```
boolean drawImage(Image img, int x, int y, ImageObserver observer)
```

Note that you have to pass an object that implements the `ImageObserver` interface to `drawImage()`. `ImageObserver` objects let you monitor the progress of image-loading operations, and we'll take a look at them later in this chapter. There's a default implementation of this interface in the `Applet` class, so we just use the `this` keyword as the `ImageObserver` object:

```
import java.awt.*;
import java.applet.*;
/* <APPLET
   CODE=ImageDemo.class
   WIDTH=350
   HEIGHT=300>
</APPLET> */
```

```
public class ImageDemo extends Applet {
Image img;
public void init() {
   img = getImage(getDocumentBase(), "Java8.jpg"); }
public void paint(Graphics g) { g.drawImage(img, 40, 40, this); }
}
```

In the `init()` method, the image returned by `getImage()` is assigned to the `image` variable. The `getImage()` uses the string "`Java8.jpg`" as the filename for the image. The image loads from a `URL` relative to the result of `getDocumentBase()`. This `URL` is actually the `URL` of the HTML page this applet tag was in.



Figure 9.4: Displaying an image

The result appears in Figure 9.4, where you can see the loaded image. Of course, there's a lot more you can do with images. For example, you can resize them (see the next solution).

Resizing Images

The Novice Programmer is working on a graphics program and needs your help. "I want to let the user resize images," the NP says. "How the heck does that work?" "No problem," you say, "you just specify the new height and width of the image in the `drawImage()` method."

To resize an image, you can use this overloaded version of the `Graphics` class's `drawImage()` method, which lets you specify an image's width and height:

```
drawImage(Image img, int x, int y, int width, int height,
ImageObserver observer)
```

The `ImageObserver` object allows you to monitor an image while it loads. Using this object you can perform other actions, such as show a progress indicator to inform you of the download progress. This kind of notification helps very much when an image is being loaded over the network.

Here's an example where you just need to press the mouse at one point and release it at another; the applet will draw the image you saw in the previous solution, resize it to fit the rectangle you have defined:

```
import java.awt.*;
import java.lang.Math;
import java.awt.event.*;
import java.applet.Applet;
/* <APPLET
   CODE=ResizeImageDemo.class
   WIDTH=300
   HEIGHT=300 >
</APPLET> */
public class ResizeImageDemo extends Applet implements MouseListener {
Image img;
boolean mouseUp = false;
Point start, end;
public void init() {
```

```
    img = getImage(getDocumentBase(), "Java8.jpg");
    addMouseListener(this); }
    public void mousePressed(MouseEvent e) {
    mouseUp = false;
    start = new Point(e.getX(), e.getY()); }
    public void mouseReleased(MouseEvent e) {
    mouseUp = true;
    end = new Point(Math.max(e.getX(), start.x),
    Math.max(e.getY(), start.y));
    start = new Point(Math.min(e.getX(), start.x),
    Math.min(e.getY(), start.y));
    repaint(); }
    public void mouseClicked(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
    public void paint (Graphics g) {
    if(mouseUp) {
    int width = end.x - start.x;
    int height = end.y - start.y;
    g.drawImage(img, start.x, start.y, width, height, this); }
    }
    }
```



Figure 9.5: Resizing an image

The result of this code appears in Figure 9.5, where you can see that we have increased the size of the image from the previous solution. We have used a few variations of the handy `drawImage()` method, which is part of the `Graphics` class. Now it's time to take a look at the `Graphics` class itself and all it contains (which is a truckload of methods). See the next solution for the details.

Drawing Graphics

The Big Boss appears in a puff of cigar smoke and says, "The design team has come up with a winner program that I want you to write." "What does it do?" you ask. "It lets the user draw lines, rectangles, and ovals as well as freehand with the mouse," the BB says. "Real cutting-edge stuff," you reply. The real core of AWT graphics is the huge AWT `Graphics` class, which is derived directly from `java.lang.Object`.

You'll find the constructor of this class in Table 9.20 and its methods in Table 9.21.

Table 9.20: The constructor of the Graphics class

Constructor	Does this
`protected Graphics()`	It creates a new `Graphics` object

Table 9.21: Methods of the Graphics class

Method	Does this
`abstract void clearRect(int x, int y, int width, height)`	It clears the given rectangle by filling it with the background color of the current drawing
`abstract void clipRect (int x, int y, int width, int height)`	It intersects the current clip region with the given rectangle
`abstract void copyArea (int x, int y, int width, int height, int dx, int dy)`	It copies an area of the component as specified by a region given by `dx` and `dy`
`abstract Graphics create()`	It creates a new `Graphics` object that's a copy of this one
`Graphics create(int x, int y, int width, int height)`	It creates a new `Graphics` object based on this one, but with a new translation and clip area
`abstract void dispose()`	It disposes of this graphics context and releases any used system resources
`void draw3DRect(int x, int y, int width, int height, boolean raised)`	It draws a 3D highlighted outline of the given rectangle
`abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)`	It draws the outline of a circular or elliptical arc covering the given rectangle
`void drawBytes (byte[] data, int offset, int length, int x, int y)`	It draws the text indicated by the given byte array
`void drawChars (char[] data, int offset, int length, int x, int y)`	It shows the text given by the given character array
`abstract boolean drawImage (Image img, int x, int y, Color bgcolor, ImageObserver observer)`	It draws as much of the given image as is available
`abstract boolean drawImage (Image img, int x, int y, ImageObserver observer)`	It dras as much of the given image as is available
`abstract boolean drawImage(Image img, int x, int y, int width, int height, Color bgcolor, ImageObserver observer)`	It draws as much of the given image as has already been scaled to fit inside the given rectangle
`abstract boolean drawImage (Image img, int x, int y, int width, int height, ImageObserver observer)`	It draws as much of the given image as has already been scaled to fit inside the given rectangle
`abstract boolean drawImage (Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, Color bgcolor, ImageObserver observer)`	It draws as much of the indicated area of the given image as is currently available, scaling it to fit inside the given area
`abstract boolean drawImage (Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, ImageObserver observer)`	It draws as much of the indicated area of the given image as is currently available, scaling it to fit inside the given area
`abstract void drawLine (int x1, int y1, int x2, int y2)`	It draws a line, using the current color, between the points `(x1, y1)` and `(x2, y2)`

Table 9.21: Methods of the Graphics class

Method	Does this
`abstract void drawOval(int x, int y, int width, int height)`	It draws the outline of an oval
`abstract void drawPolygon (int[] xPoints, int[] yPoints, int nPoints)`	It draws a closed polygon defined by arrays of `x` and `y` coordinates
`void drawPolygon (Polygon p)`	It draws the outline of a polygon defined by the given `Polygon` object
`abstract void drawPolyline (int[] xPoints, int[] yPoints, int nPoints)`	It draws a sequence of connected lines defined by arrays of `x` and `y` coordinates
`void drawRect(int x, int y, int width, int height)`	It draws the outline of the given rectangle
`abstract void drawRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)`	It draws an outlined rounded-cornered rectangle
`abstract void drawString (AttributedCharacterIterator iterator, int x, int y)`	It draws the text indicated by the given iterator applying its attributes in accordance with the specification of `TextAttribute` class
`abstract void drawString (String str, int x, int y)`	It draws the text indicated by the given string
`void fill3DRect(int x, int y, int width, int height, boolean raised)`	It paints a 3D highlighted rectangle filled with the current color
`abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)`	It fills a circular or elliptical arc covering the given rectangle
`abstract void fillOval(int x, int y, int width, int height)`	It fills an oval bounded by the given rectangle with the current color
`abstract void fillPolygon (int[] xPoints, int[] yPoints, int nPoints)`	It fills a closed polygon defined by arrays of `x` and `y` coordinates
`void fillPolygon (Polygon p)`	It fills the polygon defined by the given `Polygon` object with the graphics context's current color
`abstract void fillRect (int x, int y, int width, int height)`	It fills the given rectangle
`abstract void fillRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)`	It fills the given rounded-corner rectangle
`void finalize()`	It handles garbage collection of this graphics context
`abstract Shape getClip()`	It gets the current clipping area
`abstract Rectangle getClipBounds()`	It gets the bounding rectangle of the current clipping area
`Rectangle getClipBounds (Rectangle r)`	It gets the bounding rectangle of the current clipping area
`Rectangle getClipRect()`	It is deprecated and replaced by `getClipBounds()`
`abstract Color getColor()`	It gets this graphics context's current color
`abstract Font getFont()`	It gets the current font
`FontMetrics getFontMetrics()`	It gets the font metrics of the current font

Table 9.21: Methods of the Graphics class	
Method	**Does this**
`abstract FontMetrics getFontMetrics(Font f)`	It gets the font metrics for the given font
`boolean hitClip(int x, int y, int width, int height)`	It returns True if the rectangular area intersects the current clipping area
`abstract void setClip(int x, int y, int width, int height)`	It sets the current clip to rectangle with given coordinates
`abstract void setClip (Shape clip)`	It sets the current clipping area to an arbitrary shape
`abstract void setColor(Color c)`	It sets this graphics context's color to the given color
`abstract void setFont (Font font)`	It sets this graphics context's font to the given font
`abstract void setPaintMode()`	It sets the paint mode of the graphics context to overwrite the destination with this graphics context's current color
`abstract void setXORMode (Color c1)`	It sets the paint mode of the graphics context to alternate between this graphics context's current color and the new given color
`String to String()`	It gets a `String` object representing this `Graphics` object
`abstract void translate (int x, int y)`	It translates the origin of the graphics context to the point `(x,y)`

We are going to put the `Graphics` class to work here by creating the program the Big Boss wanted—a graphics program that lets the user draw lines, ovals, rectangles, rounded rectangles, and freehand with the mouse, as shown in Figure 9.6.

The user first picks the color of drawing out of red, gray, or orange by clicking the Red, Gray or Orange buttons. If no color button is clicked, the drawing takes place in black color. Next, the user clicks a button indicating what kind of figure he/she wants to draw, which sets a Boolean flag inside the program. When the user presses the mouse in the drawing area, that location is stored as start, using a Java `Point` object (which has two data members: `x` and `y`). When the user releases the mouse in a new location, that location is stored as end. Releasing the mouse also repaints the program, and the user can select what figure to draw—a line, oval, rectangle, or rounded rectangle—between start and end based on the Boolean flags set by clicking the buttons.

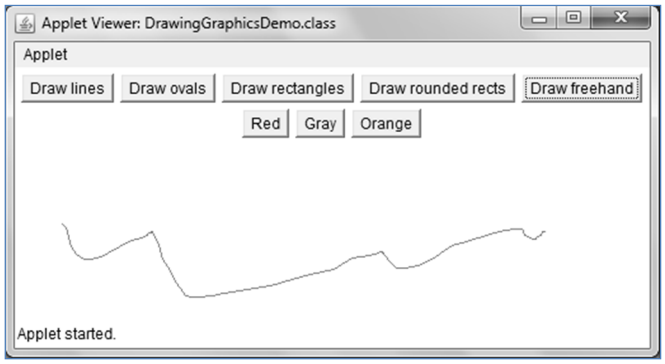


Figure 9.6: Drawing freehand with the mouse

Drawing freehand with the mouse is a little different, though. In that case, we store up to 1,000 points that the mouse moves over. When it's time to draw the figure, we just connect the dots with lines (note that a mouse event is not generated for each pixel the mouse moves over, so we need to draw lines between the mouse locations that Java does report). Here's what `DrawingGraphicsDemo`.

java looks like (we'll take a look at the drawing sections in detail in the following pages):

```
import java.awt.*;
import java.lang.Math;
import java.awt.event.*;
import java.awt.Graphics;
import java.applet.Applet;
/* <APPLET
   CODE=DrawingGraphicsDemo.class
   WIDTH=500
   HEIGHT=200 >
</APPLET> */
public class DrawingGraphicsDemo extends Applet implements ActionListener,
   MouseListener,MouseMotionListener {
   Button freeDraw, lineDraw, ovalDraw, rectDraw, roundedDraw;
   Button button1, button2, button3;
   String color="";
   Point dot[] = new Point[1000];
   Point start, end;
   int dots = 0;
   boolean mouseUp = false;
   boolean draw = false;
   boolean line = false;
   boolean oval = false;
   boolean rectangle = false;
   boolean rounded = false;
public void init() {
         lineDraw = new Button("Draw lines");
         ovalDraw = new Button("Draw ovals");
         rectDraw = new Button("Draw rectangles");
         roundedDraw = new Button("Draw rounded rects");
         freeDraw = new Button("Draw freehand");
         button1 = new Button("Red");
         button2 = new Button("Gray");
         button3 = new Button("Orange");
         add(lineDraw);
         add(ovalDraw);
         add(rectDraw);
         add(roundedDraw);
   add(freeDraw);
   add(button1);
   add(button2);
   add(button3);
         lineDraw.addActionListener(this);
         ovalDraw.addActionListener(this);
         rectDraw.addActionListener(this);
         roundedDraw.addActionListener(this);
         freeDraw.addActionListener(this);
         button1.addActionListener(this);
         button2.addActionListener(this);
         button3.addActionListener(this);
                 addMouseListener(this);
         addMouseMotionListener(this); }
public void mousePressed(MouseEvent e) {
         mouseUp = false;
         start = new Point(e.getX(), e.getY()); }
public void mouseReleased(MouseEvent e) {
         if(line) { end = new Point(e.getX(), e.getY()); }
         else {
         end = new Point(Math.max(e.getX(), start.x),
         Math.max(e.getY(), start.y));
         start = new Point(Math.min(e.getX(), start.x),
         Math.min(e.getY(), start.y)); }
         mouseUp = true;
         repaint(); }
public void mouseDragged(MouseEvent e) {
```

```
            if(draw) {
            dot[dots] = new Point(e.getX(), e.getY());
            dots++;
            repaint(); }
    }
    public void mouseClicked(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
    public void mouseMoved(MouseEvent e) { }
    public void paint (Graphics g) {
            if(color.equals("red"))g.setColor(Color.red);
            if(color.equals("gray"))g.setColor(Color.gray);
            if(color.equals("orange"))g.setColor(Color.orange);
            if (mouseUp) {
            int width = end.x - start.x;
            int height = end.y - start.y;
            if(line) {
                    g.drawLine(start.x, start.y, end.x, end.y); }
            else if(oval) {
                    g.drawOval(start.x, start.y, width, height); }
            else if(rectangle) {
                    g.drawRect(start.x, start.y, width, height); }
            else if(rounded) {
                    g.drawRoundRect(start.x, start.y, width, height, 10, 10); }
            else if(draw) {
    for(int loop_index = 0; loop_index < dots - 1;loop_index++) {
    g.drawLine(dot[loop_index].x, dot[loop_index].y,dot[loop_index + 1].x,  dot[loop_index  +
1].y); }
    }
    }
    }
    public void actionPerformed(ActionEvent e) {
            setFlagsFalse();
            if(e.getSource() == freeDraw)draw = true;
            if(e.getSource() == lineDraw)line = true;
            if(e.getSource() == ovalDraw)oval = true;
            if(e.getSource() == rectDraw)rectangle = true;
            if(e.getSource() == roundedDraw)rounded = true;
            if(e.getSource() == button1)color="red";
            if(e.getSource() == button2)color="gray";
            if(e.getSource() == button3)color="orange";}
    void setFlagsFalse() {
            rounded = false;
            line = false;
            oval = false;
            rectangle = false;
            draw = false; }
    }
```

Now that you know what `DrawingGraphicsDemo.java` looks like, let's take a look at some of its drawing functions. All these drawing functions, except for the freehand drawing functions, draw a figure between the locations `start` and `end`, which the user indicates by dragging the mouse.

Drawing Lines

Using a `Graphics` object, you can draw a line between the points (`x1, y1`) and (`x2, y2`) with the `drawLine()` method:

```
    drawLine(int x1, int y1, int x2, int y2);
```

Here's how that looks in DrawingGraphicsDemo.java:

```
    g.drawLine(start.x, start.y, end.x, end.y);
```

You can see the result in `DrawingGraphicsDemo.java` in Figure 9.7.

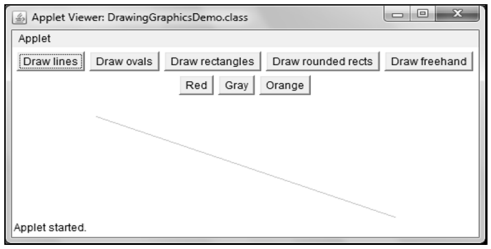


Figure 9.7: Drawing a line with the mouse

Drawing Ovals

Ellipses, including circles, are called *ovals* in AWT, and you can draw them with the `Graphics` class's `drawOval()` method:

```
drawOval(int x, int y, int width, int height);
```

Here's how ovals are drawn, as specified by the user when running `DrawingGraphicsDemo.java`:

```
int width = end.x - start.x;
int height = end.y - start.y;
g.drawOval(start.x, start.y, width, height);
```

You can see the result in `DrawingGraphicsDemo.java`, as shown in Figure 9.8.

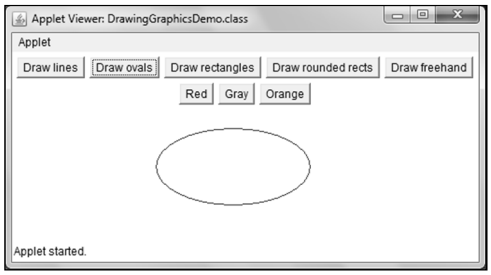


Figure 9.8: Drawing an oval with the mouse

Drawing Rectangles

You can draw rectangles using the `Graphics` class's `drawRect()` method:

```
drawRect(int x, int y, int width, int height);
```

Here's how we do it in DrawingGraphicsDemo.java:

```
int width = end.x - start.x;
int height = end.y - start.y;
g.drawRect(start.x, start.y, width, height);
```

You can see the result in `DrawingGraphicsDemo.java`, as shown in Figure 9.9.

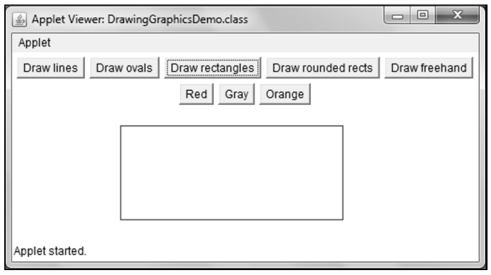


Figure 9.9: Drawing a rectangle with the mouse

Drawing Rounded Rectangles

You can draw rounded rectangles (rectangles with rounded corners that are using the `Graphics` class's `drawRoundRect()` method:

```
drawRoundRect(int x, int y, int width, int height, int arcWidth, int
arcHeight);
```

Here, you specify the arc width and height, in pixels, which specifies the rounding of the corners. Here's how we create a rounded rectangle in `DrawingGraphicsDemo.java`:

```
int width = end.x - start.x;
int height = end.y - start.y;
g.drawRoundRect(start.x, start.y, width, height, 10, 10);
```

You can see the result in `DrawingGraphicsDemo.java`, as shown in Figure 9.10:

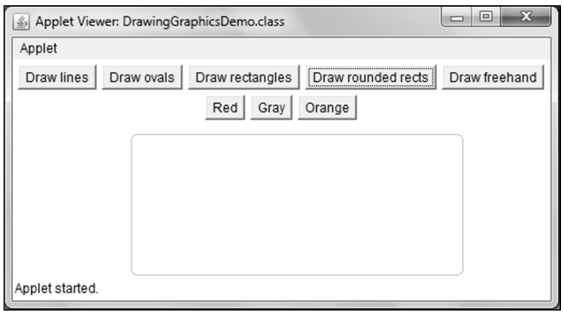


Figure 9.10: Drawing a rounded rectangle with the mouse

Drawing Freehand

You can draw freehand with the mouse using the AWT `Graphics` class, but you'll have to implement it yourself in code. Here's how we do it in `DrawingGraphicsDemo.java` using the `mouseDragged()` method. After checking to make sure the user is drawing freehand by making sure the `draw` flag is true, we save all the mouse locations in an array named `dot[]` as the mouse is dragged:

```
public void mouseDragged(MouseEvent e) {
if(draw) {
dot[dots] = new Point(e.getX(), e.getY());
dots++;
repaint(); }
}
```

Then, when it's time to draw the figure, we just connect the dots using lines, like this:

```
for(int loop_index = 0; loop_index < dots - 1; loop_index++) {
g.drawLine(dot[loop_index].x, dot[loop_index].y,
dot[loop_index + 1].x, dot[loop_index + 1].y);
}
```

You can see the result in Figure 9.6 (shown previously).

Drawing Arcs

You can use the `Graphics` class's `drawArc()` method to draw arcs (you specify angles in degrees):

```
drawArc(int x, int y, int width, int height, int startAngle, int arcAngle);
```

Drawing Polygons

There are a number of ways to draw polygons and multiple-segment lines with AWT:

```
drawPolygon(int[] xPoints, int[] yPoints, int nPoints);
drawPolygon(Polygon p);
drawPolyline(int[] xPoints, int[] yPoints, int nPoints);
```

Setting Drawing Modes

AWT also allows you to alternate between two drawing modes—straight painting mode and XOR mode—with these methods:

```
setXORMode(Color c1);
setPaintMode();
```

In painting mode, whatever you paint just covers what's underneath; but in XOR mode, what you draw will be exclusively ORed with a particular color that's already on the screen. This is very useful, because when you XOR A with B twice, B is restored, which means you can draw something on the screen and then draw it using XOR drawing. Then, whatever was on the screen originally is restored. For example, you may want to let the user "stretch" the figures he/she is drawing in DrawingGraphicsDemo.java interactively. Here's how to do this: You let the user draw a figure; then, when the user moves the mouse, you redraw the figure using XOR mode to erase it. Then you redraw it with its new size.

Selecting Colors

"My graphics work has been a little drab," says the Novice Programmer, "because everything's appearing in one color—black." "Well," you say, "you can select the drawing color easily enough." No discussion of graphics would be complete without discussing colors, and you handle colors in AWT with the Color class. You'll find the fields of this class in Table 9.22, its constructors in Table 9.23, and its methods in Table 9.24.

Table 9.22: Fields of the Color class

static Color black	static Color blue	static Color cyan
static Color darkGray	static Color gray	static Color green
static Color lightGray	static Color magenta	static Color orange
static Color pink	static Color red	static Color white
static Color yellow	static Color BLACK	static Color BLUE
static Color CYAN	static Color DARK_GRAY	static Color GRAY
static Color GREEN	static Color LIGHT_GRAY	static Color MAGENTA
static Color ORANGE	static Color PINK	static Color RED
static Color WHITE	static Color YELLOW	

Table 9.23: Constructors of the Color class

Constructor	Does this
Color(ColorSpace cspace, float[] components, float alpha)	It creates a color in the color space of the supplied ColorSpace with the color components specified in the float array and alpha
Color(float r, float g, float b)	It creates an opaque color with the given red, green, and blue values in the range 0.0 to 1.0
Color(float r, float g, float b, float a)	It creates a color with the given red, green, blue, and alpha values in the range 0.0 to 1.0
Color(int rgb)	It creates an opaque color with the given combined RGB value, consisting of the red value in bits 16 to 23, the green value in bits 8 to 15, and the blue value in bits 0 to 7
Color(int rgba, boolean hasalpha)	It creates a color with the given combined RGBA value, consisting of the alpha value in bits 24 to 31, the red value in bits 16 to 23, the green value in bits 8 to 15, and the blue value in bits 0 to 7
Color(int r, int g, int b)	It creates an opaque color with the given red, green, and blue values in the range 0 to 255
Color(int r, int g, int b, int a)	It creates a color with the given red, green, blue, and alpha values in the range 0 to 255

Table 9.24: Methods of the Color class

Method Does	Does this
`Color brighter()`	It makes a brighter version of the color
`PaintContext createContext(ColorModel cm, Rectangle r, Rectangle2D r2d, AffineTransform xform,RenderingHints hints)`	It creates and returns a `PaintContext` to generate a field pattern in solid color
`Color darker()`	It makes a darker version of the color
`static Color decode (String nm)`	It converts a string to an integer and returns the specified color
`boolean equals(Object obj)`	It determines whether another color is equal to this color
`int getAlpha()`	It gets the alpha value in the range 0 to 255
`int getBlue()`	It gets the blue value in the range 0 to 255
`static Color getColor (String nm)`	It finds a color in the system properties
`static Color getColor (String nm, Color v)`	It finds a color in the system properties
`static Color getColor (String nm, int v)`	It finds a color in the system properties
`float[] getColorComponents (ColorSpace cspace, float[] compArray)`	It gets a float array containing the color components of the `Color` object in the given color space
`float[] getColorComponents (float[]compArray)`	It gets a float array containing the color components (no alpha) of the `Color` object
`ColorSpace getColorSpace()`	It gets the color space of the `Color` object
`float[] getComponents (ColorSpace cspace, float[] compArray)`	It gets a float array containing the color and alpha components of the `Color` object in the given color space
`float[] getComponents (float[] compArray)`	It gets a float array containing the color and alpha components of the `Color` object
`int getGreen()`	It gets the green value
`static Color getHSBColor (float h, float s, float b)`	It creates a `Color` object based on the given HSB values
`int getRed()`	It gets the red value
`int getRGB()`	It gets the RGB value representing the color in the default RGB color model
`float[] getRGBColorComponents (float[] compArray)`	It gets a float array containing the color components of the `Color` object in the default color space
`float[] getRGBComponents (float[] compArray)`	It gets a float array containing the color and alpha components of the `Color` object, as represented in the default color space
`int getTransparency()`	It returns the transparency mode for this color
`int hashCode()`	It computes the hashcode for this color
`static int HSBtoRGB(float hue, float saturation, float brightness)`	It converts the components of a color, as given by the HSB model, to an equivalent set of values for the default RGB model
`static float[] RGBtoHSB (int r, int g, int b, float[] hsbvals)`	It converts the components of a color, as given by the RGB model, to an equivalent set of values for the HSB model
`String toString()`	It gets a string representation of this color

For example, to create a new color, you can specify red, green, and blue values in the `Color` class's constructor, which is shown as:

```
Color c = new Color(red, green, blue);
```

Note that the color values are specified in the range 0 to 255. Now you are free to set this new color as the drawing color by using `setForeground()`:

```
setForeground(c);
```

You can also use predefined colors, such as `Color.blue`, `Color.red`, and `Color.orange` to set the background color:

```
setBackground(Color.blue);
```

You can also fill figures with the color you have specified using `Graphics` methods, such as `fillArc()`, `fillOval()`, and so on.

The following applet shows various colors and their usage.

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/* <APPLET
  CODE=ColorsDemo.class
  WIDTH=300
  HEIGHT=200 >
</APPLET> */
public class ColorsDemo extends Applet implements MouseListener{
TextField text1;
Label label1, label2, label3, label4, label5, label6, label7, label8;
Panel panel1;
  public void init()
  {
        text1 = new TextField();
        panel1 = new Panel();
        label1 = new Label();
        label1.setBackground(Color.red);
        label1.addMouseListener(this);
        label2 = new Label();
        label2.setBackground(Color.green);
        label2.addMouseListener(this);
        label3 = new Label();
        label3.setBackground(Color.cyan);
        label3.addMouseListener(this);
        label4 = new Label();
        label4.setBackground(Color.pink);
        label4.addMouseListener(this);
        label5 = new Label();
        label5.setBackground(Color.yellow);
        label5.addMouseListener(this);
        label6 = new Label();
        label6.setBackground(Color.magenta);
        label6.addMouseListener(this);
        label7 = new Label();
        label7.setBackground(Color.orange);
        label7.addMouseListener(this);
        label8 = new Label();
        label8.setBackground(Color.blue);
        label8.addMouseListener(this);
        panel1.setLayout(new GridLayout(1,8));
        panel1.add(label1);
        panel1.add(label2);
        panel1.add(label3);
        panel1.add(label4);
        panel1.add(label5);
        panel1.add(label6);
        panel1.add(label7);
        panel1.add(label8);
        setLayout(new BorderLayout());
        add(text1,BorderLayout.CENTER);
        add(panel1,BorderLayout.NORTH);
  }
```

```
      public void mouseClicked(MouseEvent e) {
            if(e.getSource() == label1)text1.setForeground(Color.red);
            if(e.getSource() == label2)text1.setForeground(Color.green);
            if(e.getSource() == label3)text1.setForeground(Color.cyan);
            if(e.getSource() == label4)text1.setForeground(Color.pink);
            if(e.getSource() == label5)text1.setForeground(Color.yellow);
            if(e.getSource() == label6)text1.setForeground(Color.magenta);
            if(e.getSource() == label7)text1.setForeground(Color.orange);
            if(e.getSource() == label8)text1.setForeground(Color.blue);
       }
      public void mousePressed(MouseEvent e) { }
      public void mouseReleased(MouseEvent e) { }
      public void mouseEntered(MouseEvent e) { }
      public void mouseExited(MouseEvent e) { }
   }
```

The output of the program is shown here in Figure 9.11. You can see the color of the text in the text field varies according to the color selected from the displayed colors:

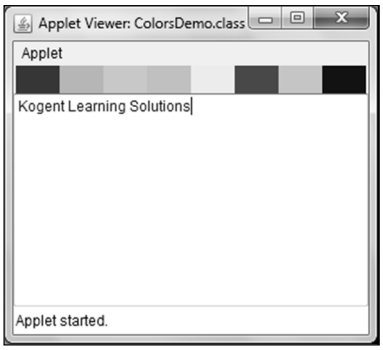


Figure 9.11: Changing color of the text

Using Canvases

The canvas component is specially built to support graphics operations. As the name suggests, canvases provide you with a blank space for you to draw on, using the `Graphics` object passed to the `paint()` method. Here's the inheritance diagram for the AWT `Canvas` class:

```
java.lang.Object
   |____java.awt.Component
         |____java.awt.Canvas
```

You'll find the constructors for this class in Table 9.25 and its methods in Table 9.26.

Table 9.25: Constructors of the Canvas class

Constructor	Does this
`Canvas()`	It creates a new canvas
`Canvas (GraphicsConfiguration config)`	It creates a new canvas given a `GraphicsConfiguration` object

Table 9.26: Methods of the Canvas class

Method	Does this
`void addNotify()`	It creates the peer of the canvas
`void createBufferStrategy (int numBuffers)`	It creates a new buffer strategy to support multi-buffering on this component
`void createBufferStrategy (int numBuffers, BufferCapabilities caps)`	It creates a new buffer strategy to support multi-buffering on this component with the required buffer capabilities

Table 9.26: Methods of the Canvas class	
Method	**Does this**
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with the canvas
`BufferStrategy getBufferStrategy()`	It returns the `BufferStrategy` used by this component
`void paint(Graphics g)`	It paints this canvas
`void update(Graphics g)`	It updates this canvas

One common use for canvases is to support an elementary form of animation, because you can use the AWT `Component` class's `setLocation()` method to move a canvas (or any other component) around. Here's an applet that does just that when it's clicked:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET
   CODE=CanvasDemo.class
   WIDTH=300
   HEIGHT=200 >
</APPLET> */
public class CanvasDemo extends Applet implements MouseListener {
DrawingCanvas dc;
Button button1;
   public void init() {
          dc = new DrawingCanvas();
          dc.setSize(260,190);
          add(dc);
          dc.addMouseListener(this); }
          public void mousePressed(MouseEvent e) {}
   public void mouseClicked(MouseEvent e) {
          for(int index = 0;index < 80;index++) {
          dc.setLocation(index, 0); }
   }
   public void mouseReleased(MouseEvent e) { }
   public void mouseEntered(MouseEvent e) { }
   public void mouseExited(MouseEvent e) { }
}
class DrawingCanvas extends Canvas {
   public void paint (Graphics g) {
          g.setColor(new Color(48,58,86));
          g.fillRect(50,35,150,119);
          g.setColor(new Color(178,98,83));
          g.fillRect(80,60,90,70);
    }
}
```

The result appears in Figure 9.12. When you click the the canvas, which displays a small figure, it moves to the left and then sweeps to the right. Note that because a `Graphics` object is passed in a canvas's `paint()` method, you can use any `Graphics` method in a canvas.

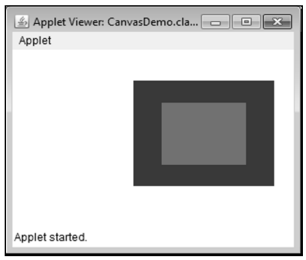


Figure 9.12: Simple animation using a canvas

Using the **ImageObserver** Interface

The Product Support Specialist appears and sadly says, "There have been complaints about how your program draws images downloaded from the Internet— only part of the image appears at first, and the rest arrives gradually." "That's the way the Internet works," you say, surprised. "Can't you fix it?" the PSS asks.

You can use the methods in the `ImageObserver` interface to watch as images are loaded; as you saw earlier, you need to indicate an `ImageObserver` interface when loading an image into an applet and specify the `Applet` class's own default implementation of that interface. In this solution, we create our own implementation of this interface. The `ImageObserver` interface has only one method, `imageUpdate()`:

```
boolean imageUpdate(Image img, int infoflags, int x, int y, int width,
int height)
```

This method is invoked whenever information about an image that's being loaded asynchronously becomes available. Here are the flags passed in the `infoflags` parameter in the `imageUpdate()` method:

- `ABORT`—It indicates that an image that was being tracked is aborted before its production was complete.
- `ALLBITS`—It indicates the completion of a static image that was drawn previously and can be redrawn in its final form.
- `ERROR`—It indicates that an image that was being tracked has encountered an error.
- `FRAMEBITS`—It indicates another complete frame that belongs to a multiframe image and is available to be redrawn presently.
- `HEIGHT`—It indicates that the height of the base image is now available (and can be read from the `height` argument of the `imageUpdate()` method).
- `PROPERTIES`—It indicates that the properties of the image are now available.
- `SOMEBITS`—It indicates that more pixels for drawing the scaled version of image are available.
- `WIDTH`—It indicates that the width of the base image is now available (and can be read from the `width` argument of the `imageUpdate()` method).

The `imageUpdate()` method returns True if further updates are required and False if you have received the information you want. Here's an example in which we just override the `imageUpdate()` method to call `repaint` to display an image, but only when it's completely loaded:

```
import java.awt.*;
import java.applet.*;
/* <APPLET
   CODE=ObserverDemo.class
   WIDTH=300
   HEIGHT=250 >
</APPLET> */
public class ObserverDemo extends Applet {
Image image;
public void init() { image = getImage(getDocumentBase(), "Java8.jpg"); }
public void paint(Graphics g) { g.drawImage(image, 20, 20, this); }
public boolean imageUpdate(Image img, int flags, int x, int y,
int w, int h) {
if ((flags & ALLBITS) != 0) { repaint(x, y, w, h); }
return (flags & ALLBITS) == 0; }
}
```

The result is shown in Figure 9.13, where the image appears only after it has been fully loaded—no flickering involved at all.

Programmers have told Sun that they find the `ImageObserver` interface to be too complex, especially when dealing with multiple downloads; therefore, Sun created the `MediaTracker` class (see the next solution).



Figure 9.13: Using an ImageObserver interface

Using the **MediaTracker** Class

Earlier Java developers could neither comprehend the `ImageObserver` interface nor manage when there were multiple images to be loaded. Sun Microsystems saw the problem and then came out with a simpler solution that could allow programmers to load images synchronously—`MediaTracker`. The `MediaTracker` class provides an easy way (easier than using `ImageObserver` objects) to monitor the downloading of images. To start, you just pass the `addImage()` method an image as well as an ID you want to use for that image (you can use the same ID for a number of images if you want to keep track of them in a group):

```
void addImage(Image image, int id)
void addImage(Image image, int id, int w, int h)
```

Here *image* is the image being tracked and *id* is its identification number. The *id* numbers may not be unique. In the other form, *w* and *h* represent the dimensions of the object when it is displayed.

You can check whether an image or image group has finished loading with the `checkID()` method, like this:

```
boolean checkID(int id)
boolean checkID(int id, boolean load)
```

This method returns True if the image or image group concerned has finished loading; otherwise, False is returned. You can also use the `waitForAll()` method to return only when all the images being tracked have been loaded.

NOTE

If you use MediaTracker once and call the addImage() method on an image, its garbage collection will not take place due to the reference in MediaTracker. Therefore, if you want the garbage collection of tracked images, ensure that the collector can collect the MediaTracker instance as well.

You'll find the fields of the `MediaTracker` class in Table 9.27, its constructor in Table 9.28, and its methods in Table 9.29.

Table 9.27: Fields of the MediaTracker class

Field	Does this
`static int ABORTED`	It indicates that the downloading of media was aborted
`static int COMPLETE`	It indicates that the downloading of media was completed
`static int ERRORED`	It indicates that the downloading of media encountered an error
`static int LOADING`	It indicates media is currently being loaded

Table 9.28: The constructor of the MediaTracker class

Constructor	Does this
`MediaTracker(Component comp)`	It creates a media tracker to track images for a given component

Table 9.29: Methods of the MediaTracker class

Method	Does this
`void addImage (Image image, int id)`	It adds an image to the images being tracked by this media tracker
`void addImage(Image image, int id, int w, int h)`	It adds a scaled image to the list of images being tracked by this media tracker
`boolean checkAll()`	It checks to see whether all images being tracked by this media tracker have finished loading
`boolean checkAll (boolean load)`	It checks to see whether all images tracked by this media tracker have finished loading
`boolean checkID(int id)`	It checks to see whether all images tracked by this media tracker that are tagged with the given identifier have finished loading
`boolean checkID (int id, boolean load)`	It checks to see whether all images tracked by this media tracker that are tagged with the given identifier have finished loading
`Object[] getErrorsAny()`	It gets a list of media that have encountered an error
`Object[] getErrorsID (int id)`	It gets a list of media with the given ID that have encountered an error
`boolean isErrorAny()`	It checks the error status of all the images
`boolean isErrorID(int id)`	It checks the error status of all the images tracked by this media tracker with the given identifier
`void removeImage (Image image)`	It removes the given image from this media tracker
`void removeImage (Image image, int id)`	It removes the given image with the given tracking ID from this media tracker
`void removeImage (Image image, int id, int width, int height)`	It removes the given image with the given width, height, and ID from this media tracker
`int statusAll(boolean load)`	It returns the bitwise inclusive OR of the status of all media tracked by this media tracker
`int statusID (int id, boolean load)`	It returns the bitwise inclusive OR of the status of all media with the given identifier that are tracked by this media tracker
`void waitForAll()`	It starts loading all images tracked by this media tracker
`boolean waitForAll(long ms)`	It starts loading all images tracked by this media tracker
`void waitForID(int id)`	It starts loading all images tracked by this media tracker with the given identifier
`boolean waitForID(int id, long ms)`	It starts loading all images tracked by this media tracker with the given identifier

Here's an example. In this case, we use the `waitForAll()` method to wait until an image is fully loaded (note the `try/catch` statement, which is there to handle exceptions —you'll see this in more detail later in this book):

```
import java.awt.*;
import java.applet.*;
/* <APPLET
  CODE=TrackMedia.class
  WIDTH=300
  HEIGHT=250 >
</APPLET> */
public class TrackMedia extends Applet {
Image image;
public void init() {
MediaTracker tracker = new MediaTracker(this);
image = getImage(getDocumentBase(), "Java8.jpg");
tracker.addImage(image, 0);
try { tracker.waitForAll(); }
catch (InterruptedException ex) { }
```

```
    }
    public void paint(Graphics g) { g.drawImage(image, 20, 20, this); }
    }
```

The results appear in Figure 9.14.



Figure 9.14: Using a MediaTracker

Working Pixel by Pixel: The **PixelGrabber** and **MemoryImageSource** Classes

"OK," says the Novice Programmer, "there are few things I want to do graphically that I can only do by gaining direct access to the pixels in an image. Guess I can't use Java for that." "Of course you can," you say. The NP says, "Tell me more!"

You can use `PixelGrabber` objects to place the pixels from an image into an array and thus gain direct access to those pixels. In other words, it is the inverse of the `MemoryImageSource`. While a `MemoryImageSource` class creates a new image from an array of data, a `PixelGrabber` takes an existing image and grabs the pixel array from it.

Here's the inheritance diagram for the `PixelGrabber` class:

```
java.lang.Object
  |____java.awt.image.PixelGrabber
```

You'll find the constructors for the `PixelGrabber` class in Table 9.30 and the methods for this class in Table 9.31.

Table 9.30: Constructors of the PixelGrabber class

Constructor	Does this
`PixelGrabber(Image img, int x, int y, int w, int h, boolean forceRGB)`	It creates a `PixelGrabber` object to grab the `(x, y, w, h)` section of pixel from the given image
`PixelGrabber(Image img, int x, int y, int w, int h, int[] pix, int off, int scansize)`	It creates a `PixelGrabber` object to grab the `(x, y, w, h)` section of pixels from the given image and places them in an array
`PixelGrabber(ImageProducer ip, int x, int y, int w, int h, int[] pix, int off, int scansize)`	It creates a `PixelGrabber` object to grab the `(x, y, w, h)` rectangular section of pixels from the image produced by the image producer into the given array

Table 9.31: Methods of the PixelGrabber class

Method	Does this
`void abortGrabbing()`	It asks the `PixelGrabber` object to abort the image fetch

Table 9.31: Methods of the PixelGrabber class	
Method	**Does this**
`ColorModel getColorModel()`	It gets the color model for the pixels in the array
`int getHeight()`	It gets the height of the pixel buffer (after adjusting for image height)
`Object getPixels()`	It gets the pixel buffer
`int getStatus()`	It gets the status of the pixels
`int getWidth()`	It gets the width of the pixel buffer (after adjusting for image width)
`boolean grabPixels()`	It requests the image or image producer to start delivering pixels and wait for all of the pixels in the rectangle of interest to be delivered
`boolean grabPixels(long ms)`	It requests the image or image producer to start delivering pixels and wait for all the pixels in the rectangle of interest to be delivered or wait until the given timeout has elapsed
`void imageComplete (int status)`	This is part of the `ImageConsumer` API, which this class must implement to retrieve the pixels
`void setColorModel (ColorModel model)`	This is part of the `ImageConsumer` API, which this class must implement to retrieve the pixels
`void setDimensions (int width, int height)`	This is part of the `ImageConsumer` API, which this class must implement to retrieve the pixels
`void setHints(int hints)`	This is part of the `ImageConsumer` API, which this class must implement to retrieve the pixels
`void setPixels(int srcX, int srcY, int srcW, int srcH, ColorModel model, byte[] pixels, int srcOff, int srcScan)`	This is part of the `ImageConsumer` API, which this class must implement to retrieve the pixels
`void setPixels(int srcX, int srcY, int srcW, int srcH, ColorModel model, int[] pixels, int srcOff, int srcScan)`	This is part of the `ImageConsumer` API, which this class must implement to retrieve the pixels
`void setProperties (Hashtable<?,?> props)`	This is part of the `ImageConsumer` API, which this class must implement to retrieve the pixels
`void startGrabbing()`	It asks the `PixelGrabber` to start fetching pixels
`int status()`	It returns the status of the pixels

To create an image from an array of pixels, you can use the `MemoryImageSource` class. The inheritance diagram for this class looks like this:

```
java.lang.Object
  |____java.awt.image.MemoryImageSource
```

You'll find the constructors for the `MemoryImageSource` class in Table 9.32 and the methods for this class in Table 9.33.

Table 9.32: Constructors of the MemoryImageSource class	
Constructor	**Does this**
`MemoryImageSource(int w, int h ColorModel cm, byte[] pix, int off, int scan)`	It creates an `ImageProducer` object that uses an array of bytes to produce the data for an `Image` object
`MemoryImageSource(int w, int h, ColorModel cm, byte[] pix, int off, int scan, Hashtable<?,?> props)`	It creates an `ImageProducer` object that uses an array of bytes to produce the data for an `Image` object

Table 9.32: Constructors of the MemoryImageSource class	
Constructor	**Does this**
`MemoryImageSource(int w, int h, ColorModel cm,int[] pix, int off, int scan)`	It creates an `ImageProducer` object that uses an array of integers to produce the data for an `Image` object
`MemoryImageSource(int w, int h, ColorModel cm, int[] pix, int off, int scan, Hashtable<?,?> props)`	It creates an `ImageProducer` object that uses an array of integers to produce the data for an `Image` object
`MemoryImageSource(int w, int h, int[] pix, int off, int scan)`	It creates an `ImageProducer` object that uses an array of integers in the default RGB color model to produce the data for an `Image` object
`MemoryImageSource(int w, int h, int[] pix, int off, int scan, Hashtable<?,?> props)`	It creates an `ImageProducer` object that uses an array of integers in the default RGB color model to produce the data for an `Image` object

Table 9.33: Methods of the MemoryImageSource class	
Method	**Does this**
`void addConsumer (ImageConsumer ic)`	It adds an `ImageConsumer` object to the list of consumers interested in data for this image
`boolean isConsumer (ImageConsumer ic)`	It indicates whether an `ImageConsumer` object is on the list of consumers interested in data for this image
`void newPixels()`	It sends a whole new buffer of pixels to any `ImageConsumer` objects interested in data for this image and notify them of completion of an animation frame
`void newPixels(byte[] newpix, ColorModel newmodel, int offset, int scansize)`	It changes to a new byte array to hold the pixels for this image
`void newPixels (int[] newpix, ColorModel newmodel, int offset, int scansize)`	It changes to a new integer array to hold the pixels for this image
`void newPixels(int x, int y, int w, int h)`	It sends a rectangular region of the buffer of pixels to `ImageConsumer` objects interested in data for this image and notify them of completion of an animation frame
`void newPixels(int x, int y, int w, int h, boolean framenotify)`	It sends a rectangular region of the buffer of pixels to `ImageConsumer` objects interested in data for this image
`void removeConsumer (ImageConsumer ic)`	It removes an `ImageConsumer` object from the list of consumers interested in data for this image
`void requestTopDownLeftRightResend(ImageConsumer ic)`	It requests that the image data be delivered one more time in top-down, left-right order
`void setAnimated (boolean animated)`	It changes this memory image into a multiframe animation or a single-frame static image
`void setFullBufferUpdates (boolean fullbuffers)`	It indicates whether this animated memory image should always be updated by sending the complete buffer of pixels whenever there is a change
`void startProduction (ImageConsumer ic)`	It adds an `ImageConsumer` object to the list of consumers and starts delivery of the image data

Here's an example using the `PixelGrabber` and `MemoryImageSource` classes. In this case, we just read in an image and copy it to a new image object. We do this by loading in the image we have used in the previous examples in this chapter, which are 230 by 218 pixels. First, we load the image into `img1` variable; place the pixels in that image into an array named `pixels` using a `PixelGrabber` object's `grabPixels()` method; and

then create a new image, `img2`, using a `MemoryImageSource` object and the `Applet` class's `createImage()` method:

```
import java.awt.*;
import java.applet.*;
import java.awt.image.*;
/* <APPLET
   CODE=CopierDemo.class
   WIDTH=300
   HEIGHT=250 >
</APPLET> */
public class CopierDemo extends Applet {
Image img1, img2;
public void init() {
img1 = getImage(getDocumentBase(), "Java8.jpg");
int pixels[] = new int[230 * 218];
PixelGrabber grabber = new PixelGrabber(img1, 0, 0, 230, 218,pixels, 0, 230);
try { grabber.grabPixels(); }
catch (InterruptedException e) {}
img2 = createImage(new MemoryImageSource(230, 218, pixels, 0,230)); }
public void paint(Graphics g) { g.drawImage(img2, 10, 10, this); }
}
```

That's all there is to it—now the image is copied into `img2` and displayed. Now that we have access to the pixels in the image, we'll do a few more things with the image, such as brightening it, as discussed in the next solution.

Brightening Images

You can brighten images just by increasing their red, green, and blue color values by the same amount. In the following code, we add 80 to each color value:

```
import java.awt.*;
import java.applet.*;
import java.awt.image.*;
/* <APPLET
   CODE=BrightDemo.class
   WIDTH=300
   HEIGHT=250 >
</APPLET> */
public class BrightDemo extends Applet {
Image img1, img2;
public void init() {
img1 = getImage(getDocumentBase(), " Java8.jpg");
int pixels[] = new int[230 * 218];
PixelGrabber grabber = new PixelGrabber(img1, 0, 0, 230,218, pixels, 0, 230);
try { grabber.grabPixels(); }
catch (InterruptedException e)  { }
for (int loop_index = 0; loop_index < 485 * 88; loop_index++) {
int p = pixels[loop_index];
int red = (0xff & (p >> 16)) + 80;
int green = (0xff & (p >> 8)) + 80;
int blue = (0xff & p) + 80;
if (red > 255) red = 255;
if (green > 255) green = 255;
if (blue > 255) blue = 255;
pixels[loop_index] = (0xff000000 | red << 16 | green << 8 |blue); }
img2 = createImage(new MemoryImageSource(230, 218, pixels, 0,230)); }
public void paint(Graphics g) { g.drawImage(img2, 10, 10, this); }
}
```

The result appears in Figure 9.15. Now we are really manipulating images in code.



Figure 9.15: Brightening Image

Converting Images to Grayscale

The Grayscale file is derived from `RGBImageFilter`. It computes the brightness of a pixel by taking as input the values of red, green, and blue colors, and returns a gray pixel, which has same brightness as the source, as output. The output produced is in accordance with the NTSC (National Television Standards Committee) color-to-brightness conversion factor.

You can also convert images to grayscale by averaging the red, green, and blue color values for each pixel. Here, we convert the image, `Java8.jpg`, to grayscale. Here's how this looks in code:

```
import java.awt.*;
import java.applet.*;
import java.awt.image.*;
/*<APPLET
  CODE=GrayscaleDemo.class
  WIDTH=300
  HEIGHT=250 >
</APPLET> */
public class GrayscaleDemo extends Applet {
Image img1, img2;
public void init() {
img1 = getImage(getDocumentBase(), "Java8.jpg");
int pixels[] = new int[230 * 218];
PixelGrabber grabber = new PixelGrabber(img1, 0, 0, 230, 218,pixels, 0, 230);
try { grabber.grabPixels(); }
catch (InterruptedException e) {}
for (int loop_index = 0; loop_index < 230 * 218; loop_index++) {
int p = pixels[loop_index];
int red = (0xff & (p >> 16));
int green = (0xff & (p >> 8));
int blue = (0xff & p);
int avg = (int) ((red + green + blue) / 3);
pixels[loop_index] = (0xff000000 | avg << 16 | avg << 8 | avg); }
img2 = createImage(new MemoryImageSource(230, 218, pixels, 0,230)); }
  public void paint(Graphics g) { g.drawImage(img2, 10, 10, this); }
}
```

The result appears in Figure 9.16. Of course, seeing a grayscale image in this figure might not be too convincing in a book full of black-and-white images.



Figure 9.16: Converting an image to grayscale

Embossing Images

Embossing is a powerful effect in which images appear to be raised from the viewing surface; we'll take a look at this effect here. Embossing images is most convenient when you work in terms of a two-dimensional array. However, the `PixelGrabber` and `MemoryImageSource` classes only work with one-dimensional arrays, so we'll simulate two-dimensional addressing by multiplying and adding array indexes in this example. Here, we emboss the `Java8.jpg` image you saw in the previous solution:

```
    import java.awt.*;
    import java.awt.image.*;
    import java.applet.*;
    /*<APPLET
      CODE=EmbossDemo.class
      WIDTH=300
      HEIGHT=300 >
    </APPLET> */
    public class EmbossDemo extends Applet {
      Image img1, img2;
      public void init() {
            img1 = getImage(getDocumentBase(), "Java8.jpg");
            int pixels[] = new int[248 * 248];
    PixelGrabber grabber = new PixelGrabber(img1, 0, 0, 248, 248,pixels, 0, 248);
      try { grabber.grabPixels(); }
      catch (InterruptedException e)  { }
      for (int x = 2; x < 247; x++) {
      for (int y = 2; y < 247; y++) {
    int red = ((pixels[(x + 1) * 248 + y + 1] & 0xFF) -(pixels[x * 248 + y] & 0xFF)) + 128;
    int green = (((pixels[(x + 1) * 248 + y + 1] & 0xFF00) /0x100) % 0x100 - ((pixels[x *
      248 + y] & 0xFF00) /0x100) % 0x100) + 128;
    int blue = (((pixels[(x + 1) * 248 + y + 1] & 0xFF0000) / 0x10000) % 0x100 - ((pixels[x *
248 + y] & 0xFF0000) / 0x10000) % 0x100) + 128;
            int average = (red + green + blue) / 3;
    pixels[x * 248 + y] = (0xff000000 | average << 16 | average << 8 | average); } }
    img2 = createImage(new MemoryImageSource(248, 248, pixels, 0,248));
    }
      public void paint(Graphics g) { g.drawImage(img2, 10, 10, this); }
    }
```



Figure 9.17: Embossing an image

The result appears in Figure 9.17. As you can see, the figure does indeed appear as though it were embossed.

Summary

In this chapter, we introduced you to the concepts of graphics, images, texts, fonts, and input devices followed by their usage. You learned how to draw the graphics (lines, rectangles, or arcs) and resize the images. Later, you learned the use of `ImageObserver` interface and the `MediaTracker` class. Toward the end, the chapter helped you to develop applications to brighten the images and convert them to grayscale with embossing of those images.

The next chapter discusses about AWT- windows, menus, and dialog boxes.



10

AWT—Windows, Menus, and Dialog Boxes

If you need an immediate solution to:	*See page:*
Creating Frame Windows	318
Showing and Hiding Windows	319
Handling Window Events	320
Automatically Hiding Windows upon Closing	323
Using the Window Class	323
Creating Menus	330
Creating a MenuBar Object	331
Creating Menu Objects	332
Creating MenuItem Objects	333
Handling Menu Events	335
More Menu Options	336
Adding Menu Separators	338
Disabling Menu Items	338
Adding Checkboxes to Menus	339
Creating Submenus	341
Pop-Up Menus	342
Dialog Boxes	344
File Dialog Boxes	348

In Depth

This chapter is all about taking the next step beyond creating controls in AWT programs. In this chapter, we'll create and display AWT windows and see all that goes into the process, using the AWT `Window` and `Frame` classes. We'll also look at how to use the `Dialog` class to create dialog boxes and the `FileDialog` class to create file dialog boxes for specifying a file. This is also where we'll take a look at menus, because in AWT programming, you need a window before you can display menus. First, we'll take a brief look at all these items in overview.

Windows

Windows, of course, are the foundation of GUI programming. Just about everything having to do with the user interface in a graphical environment happens in a window, and every GUI user is familiar with windows. You host the components that make up an applet or windowed application in windows, and there are three types of windows available to AWT programmers. The first type of window is the applet window, where the `Applet` class manages the window itself, and automatically creates and handles the window. You can also create frame windows, which are what you normally think of as windows. A frame window supports a frame around a window, a title bar, minimize, maximize, and close buttons. We'll create one here using the `Frame` class.

TIP

When you launch a frame window from an applet, Java adds a warning sign on the top right side of the window for security purposes. The message "Java Applet Window" appears as a tooltip when you bring the mouse pointer over that sign.

There's another window class you can use—the `Window` class. This class just presents a blank window—no title bar, no frame, just a blank rectangle. You are responsible for displaying what you want in these windows. As you might expect, the `Frame` class is derived from the `Window` class. Paradoxically, however, you can't create a direct `Window` object in your own programs unless you already have a frame window, because the publicly available `Window` constructor requires you to pass either a `Frame` window object or another `Window` object.

Menus

Every GUI user knows about menus—they are those indispensable controls that hide all the options a program can present by category. Imagine if all the options a word processor could present were available as buttons, visible all at once— there would be no space to enter text. Menus let you store those options away in a compact way. This is a very attractive GUI technique, because space is always at a premium in windowed environments.

In AWT programming, you need a frame window to use menus. You create a `MenuBar` object and add that menu bar to a frame window with the window's `setMenuBar()` method. You also create objects of the `Menu` class to create the individual menus (such as File, Edit, and so on) that appear in the menu bar, and you create objects of the `MenuItem` class to create the actual items in each menu (such as New, Open, Help, Exit, and so on).

You can also support some nice options in menus—submenus that open when you select a menu item and checkboxes that let you toggle menu items on and off (such as Automatic Spell Checking and View Toolbar). We'll look at all this in the chapter.

Dialog Boxes

Windowed programs frequently use dialog boxes to get user input, such as the name of a file to open, a password, or a color that's selected from many colors. Like the other visual elements in this chapter, dialog boxes are familiar to nearly all GUI users. You use dialog boxes when you want to get user input but don't want to display a dedicated control, such as a text field, at all times for that option in your main window. That is, dialog boxes are temporary windows you can fill with controls for user input.

You'll see two types of dialog boxes in this chapter, as supported by the `Dialog` and `FileDialog` classes. You use the `Dialog` class as a base class for the dialog boxes you create and customize, as you'll see here. On the other hand, you don't usually need to derive a class from the `FileDialog` class—this class presents a file dialog box, which the user can use to select a file. The methods and data members of this class are sufficient for most file-selection purposes, and all you need to do is to instantiate an object of this class and use it. You'll see all this in the chapter, too.

That's it. Now you have an overview of what's in this chapter. Now, let's turn to the "Immediate Solutions" section.

Immediate Solutions

Creating Frame Windows

"OK," says the Novice Programmer, "I'm working on my word processor in Java and want to present various views of the document—each view is a window into a different place in the document. What do you think?" You say, "I think you should think about letting the user launch new frame windows."

You have already seen the `Frame` window class in this book, because you use that class as the foundation of applications in AWT programming. Here's the inheritance diagram for `Frame` class:

```
java.lang.Object
   |____java.awt.Component
         |____java.awt.Container
                |____java.awt.Window
                       |____java.awt.Frame
```

You can see the constructors and methods of this class in Chapter 6, in Tables 6.4 and 6.5 respectively. We'll create a frame window over the next few solutions, showing and hiding it as needed, giving it a size, adding controls to it, and handling events in it. To start, we'll create a new class, `FrameDemo`, based on the `Frame` class, and we'll display a label with the text "Kogent Learning Solutions" First, we declare this new class by extending the `Frame` class:

```
import java.awt.*;
class FrameDemo extends Frame { . . . }
```

Next, we create a constructor for the class. We are going to use the `Frame` class constructor, which takes a title for the window. Therefore, we add a title parameter to the constructor and pass that back to the `Frame` class, like this:

```
import java.awt.*;
class FrameDemo extends Frame {
FrameDemo(String title) {
        super(title);
. . . }
}
```

Here's an important point: The default layout manager in frame windows is the `BorderLayout` manager (unlike applet windows, which use the `FlowLayout` manager). Note that this applies for both frame windows you display from applets and frame windows you use in windowed applications. If you want to use a different layout manager, you have to set it yourself. In this case, we install a flow layout manager by using the `setLayout()` method, like this:

```
import java.awt.*;
class FrameDemo extends Frame {
FrameDemo(String title) {
super(title);
setLayout(new FlowLayout());
. . . }
}
```

Now we are ready to create a label and add it to the layout:

```
import java.awt.*;
class FrameDemo extends Frame {
Label label1;
FrameDemo(String title) {
super(title);
setLayout(new FlowLayout());
label1 = new Label("Kogent Learning Solutions!");
add(label1); }
}
```

That completes the `Frame` window class—the next step is to display it. See the following solution for details.

Showing and Hiding Windows

"I have created a `Frame` window class," the Novice Programmer says. "Now how do I display a window of that class on the screen?" "That's not so hard," you say. "After giving the window a size, you just use the `setVisible()` method."

Here's the `Frame` window class, `FrameDemo`, we developed in the previous solution:

```
import java.awt.*;
class FrameDemo extends Frame {
Label label1;
FrameDemo(String title) {
super(title);
setLayout(new FlowLayout());
label1 = new Label("Kogent Learning Solutions");
add(label1); }
}
```

We'll launch this window from an applet. To do that, we display a text field for user to enter text. Here's how we add the text field in the applet:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
/* <APPLET
        CODE=ShowHideFrame.class
        WIDTH=300
        HEIGHT=200>
</APPLET> */
public class ShowHideFrame extends Applet implements ActionListener {
TextField text1;
public void init() {
text1 = new TextField(30);
add(text1);
text1.addActionListener(this);
. . . }
}
```

Now we'll create a new object of the `FrameDemo` class, giving it the title "Java Frame ":

```
public class ShowHideFrame extends Applet implements ActionListener {
TextField text1;
FrameDemo frame;
public void init() {
text1 = new TextField(30);
add(text1);
text1.addActionListener(this);
frame = new FrameDemo("Java Frame");
. . .
}
```

Before you can display a window, you must give it a size, using the `setSize()` method; otherwise, it won't appear on the screen. Here's how we use `setSize()` to give this window the dimensions 300 by 200 pixels:

```
public void init() {
    text1 = new TextField(30);
    add(text1);
    text1.addActionListener(this);
    frame = new FrameDemo("Java Frame");
    frame.setSize(300, 200);
}
```

The new window object is ready to go. To display the window, we can use the `setVisible()` method. We pass it a value True to display the window and False to hide the window. Here's what this looks like for the two buttons in this applet:

```
public void actionPerformed(ActionEvent event) {
String text = text1.getText();
if(text.equalsIgnoreCase("show")){ frame.setVisible(true); }
else if(text.equalsIgnoreCase("hide")){ frame.setVisible(false); }
else text1.setText("Please enter either show or hide to show/hide frame");
}
```

That's all that's needed. Now the applet is ready to go, as you can see in Figure 10.1. When you enter "show" in the text field, the frame window appears, with warning sign on the top right side of the window (which is not visible when you use a frame window in an application), as you can see in the figure. When you type "hide" in the text field and press the Enter key, the window disappears from the screen.



Figure 10.1: Displaying a frame window

Handling Window Events

"Hmm," says the Novice Programmer, "I have put three hundred buttons in a frame window; but when I click them, nothing happens." You smile and say, "Do you handle button click events in your frame window?" "Uh-oh," says the NP.

You can handle events in windows just as you can handle them anywhere—by implementing the methods of `listener` interfaces or by using adapter classes. Here's an example in which we handle window events in the frame window developed over the previous two solutions by implementing the methods in the `WindowListener` interface and display appropriate message in the text field. Here's the code:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
/* <APPLET
   CODE=WindowEventDemo.class
   WIDTH=300
   HEIGHT=200>
</APPLET> */
public class WindowEventDemo extends Applet implements ActionListener{
   TextField text1;
   FrameDemo frame;
      public void init() {
          text1 = new TextField(30);
          add(text1);
          text1.addActionListener(this);
          frame = new FrameDemo("Handling Window Events",this); }
   public void actionPerformed(ActionEvent event) {
          String text = text1.getText();
          if(text.equalsIgnoreCase("show")){ frame.setVisible(true); }
          else if(text.equalsIgnoreCase("hide")){ frame.setVisible(false); }
```

```
    else text1.setText("Please enter either show or hide to show/hide frame");}
    }
    class FrameDemo extends Frame implements WindowListener {
       Label label1;
       WindowEventDemo demo;
       FrameDemo(String title, WindowEventDemo we) {
              super(title);
              demo = we;
              setLayout(new FlowLayout());
              label1 = new Label("Kogent Learning Solutions!");
              add(label1);
              addWindowListener(this);
              setSize(300,200);
       }
    public void windowDeactivated(WindowEvent e){demo.text1.setText("Window Deactivated"); }
    public void windowActivated(WindowEvent e){demo.text1.setText("Window Activated"); }
       public void windowDeiconified(WindowEvent e){}
       public void windowIconified(WindowEvent e){}
       public void windowClosed(WindowEvent e){ }
    public void windowOpened(WindowEvent e){demo.text1.setText("Window Opened"); }
       public void windowClosing(WindowEvent e){}
    }
```

The result appears in Figure 10.2. As you can see in the figure, the frame window now handles `WindowListener` events, and appropriate message is displayed in the text field. Note that we've given the body of only three methods, `windowDeactivated()`, `windowActivated()`, and `windowOpened()`. You can always provide the body of other methods too as per your requirement. That's it.

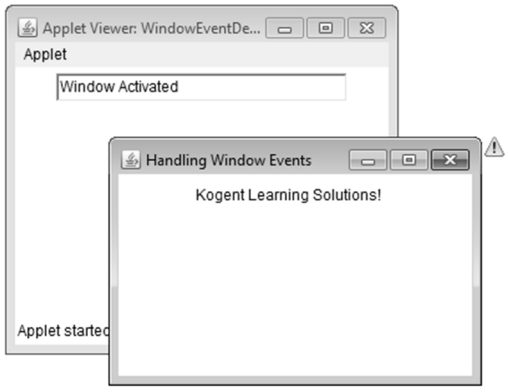


Figure 10.2: Handling window events in a frame window

There are events specific to windows—that is, there's an entire `WindowListener` interface, and you can see the methods of this interface in Table 10.1.

Table 10.1: Methods of the WindowListener interface

Methods	Does this
`void windowActivated (WindowEvent e)`	It is called when the window is set to the user's active window, which means the window will receive keyboard events
`void windowClosed (WindowEvent e)`	It is called when a window has been closed as a consequence of calling `dispose` method on the window
`void windowClosing (WindowEvent e)`	It is called when the user closes the window from the window's system menu
`void windowDeactivated (WindowEvent e)`	It is called when a window is no longer the user's active window
`void windowDeiconified (WindowEvent e)`	It is called when a window is changed from a minimized to a normal state

Table 10.1: Methods of the WindowListener interface	
Methods	**Does this**
`void windowIconified (WindowEvent e)`	It is called when a window is changed from a normal to a minimized state
`void windowOpened (WindowEvent e)`	It is called the first time a window is made visible

Note that you can also use the `WindowAdapter` class to handle window events; this class already implements all the methods of the `WindowListener` interface, and you just override the methods you want to use. The methods of the `WindowListener` interface and `WindowAdapter` class are passed an object of the `WindowEvent` class. You'll find the fields of this class in Table 10.2, its constructor in Table 10.3, and its methods in Table 10.4.

Table 10.2: Fields of the WindowEvent class	
Field	**Does this**
`static int WINDOW_ACTIVATED`	"Window activated" event
`static int WINDOW_CLOSED`	"Window closed" event
`static int WINDOW_CLOSING`	"Window closing" event
`static int WINDOW_DEACTIVATED`	"Window deactivated" event
`static int WINDOW_DEICONIFIED`	"Window deiconified" event
`static int WINDOW_FIRST`	The first number in the range of IDs used for window events
`static int WINDOW_GAINED_FOCUS`	"Window getfocus" event
`static int WINDOW_ICONIFIED`	"Window iconified" event
`static int WINDOW_LAST`	The last number in the range of IDs used for window events
`static int WINDOW_LOST_FOCUS`	"Window lostfocus" event
`static int WINDOW_OPENED`	"Window opened" event
`static int WINDOW_STATE_CHANGED`	"Window statechange" event

Table 10.3: The constructor of the WindowEvent class	
Constructor	**Does this**
`WindowEvent(Window source, int id)`	It creates a `WindowEvent` object
`WindowEvent(Window source, int id, int oldState, int newState)`	It creates a `WindowEvent` object with the specified previous and new window states
`WindowEvent(Window source, int id, Window opposite)`	It creates a `WindowEvent` object with the specified opposite `Window`
`WindowEvent(Window source, int id, Window opposite, int oldState, int newState)`	It creates a `WindowEvent` object with specified parameters

Table 10.4: Methods of the WindowsEvent class	
Method	**Does this**
`int getNewState()`	It returns the new state of the window for `WINDOW_STATE_CHANGED` event
`int getOldState()`	It returns the old state of the window for `WINDOW_STATE_CHANGED` event
`Window getOppositeWindow()`	It returns the other Window involved in this focus or activation change
`Window getWindow()`	It gets the originator of the event
`String paramString()`	It gets a parameter string identifying this event

We put one of the `WindowListener` interface methods to work—the `windowClosing` event—in the next solution.

Automatically Hiding Windows upon Closing

"Hey," says the Novice Programmer, "Java's gone all wacky again. When I try to close a frame window, nothing happens. I'm emailing Oracle about it—what a bug!" "It's not a bug," you say. "It's just up to you to close a window yourself, that's all." "Oh," says the NP, "how does that work?"

To close a frame window when the user clicks the close button in the upper-right corner of the title bar, you can handle the `windowClosing` event. Here, we are using an anonymous adapter inner class to do it (see Chapter 6 for more details). In this case, we are just hiding the window when the user clicks the close button:

```
import java.awt.*;
import java.awt.event.*;
class FrameDemo extends Frame {
Label label1;
FrameDemo(String title) {
super(title);
setLayout(new FlowLayout());
label1 = new Label("Kogent Learning Solutions!");
add(label1);
addWindowListener(new WindowAdapter() {public void
windowClosing(WindowEvent e) { setVisible(false); } });
}
}
```

Here's something useful to note: When you close a window this way, it still exists as an object, which means you can still access its methods and data members. This will be useful when we are working with dialog boxes and want to retrieve the data the user entered.

Using the **Window** Class

"I want to customize the appearance of my windows entirely," the Novice Programmer says. "I'm going to display the works of the old painting masters and want to make the frame look like a real gilt painting frame." "Hmm," you say, "sounds like a big seller. You should use the `Window` class and customize it."

You can use the `Window` class to create an entirely blank window that's ready to be customized. You'll find the constructors for the `Window` class in Table 10.5 and its methods in Table 10.6.

Table 10.5: Constructors of the Window class

Constructor	Does this
`Window(Frame owner)`	It creates a new invisible window with the given frame as its owner
`Window(Window owner)`	It creates a new invisible window with the given window as its owner
`Window(Window owner, GraphicsConfiguration gc)`	It creates a new invisible window with the given window as its owner and a graphic configuration of a screen device

Table 10.6: Methods of the Window class

Method	Does this
`void addNotify()`	It makes the window displayable by creating the connection to its native screen resource
`void addPropertyChangeListener (PropertyChangeListner listener)`	It adds a property change listener to the listener list
`void addPropertyChangeListener (String propertyName, PropertyChangeListner listener)`	It adds a property change listener to the listener list for a specific property
`void addWindowFocusListener (WindowFocusListener l)`	It adds the given window focus listener to receive window events from this window
`void addWindowListener (WindowListener l)`	It adds the given window listener to receive window events from this window

Table 10.6: Methods of the Window class

Method	Does this
`void addWindowStateListener (WindowStateListener l)`	It adds the given window state listener to receive window events from this window
`void applyResourceBundle (ResourceBundle rb)`	It is deprecated, replaced by `Component.applyComponentOrientation(ComponentOrientation)`
`void applyResourceBundle (String rbName)`	It is deprecated, replaced by `Component.applyComponentOrientation(ComponentOrientation)`
`void createBufferStrategy (int numBuffers)`	It creates a new strategy to support multi-buffering on this component
`void createBufferStrategy (int numBuffers, BufferCapabilities caps)`	It creates a new strategy to support multi-buffering on this component with the required buffer capabilities
`void dispose()`	It releases all the native screen resources used by this window, its subcomponents, and all of its owned children
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this window
`Color getBackground()`	It gets the background color of this window
`BufferStrategy getBufferStrategy()`	It returns the `BufferStrategy` used by this component
`boolean getFocusableWindowState()`	It returns True when this window can become the focused window and meet the other requirements outlined in the `isFocusableWindow`, otherwise it returns False
`Container getFocusCycleRoot Ancestor()`	It gets the returns null as windows has no ancestors (they represent the top of component hierarchy)
`Component getFocusOwner()`	It gets the child component of this window, which has focus if this window is active otherwise returns null
`Set <AWTKeyStroke> getFocusTraversalKeys (int id)`	It gets a focus traversal key for this window
`List<Image> getIconImages()`	It gets the sequence of images to be displayed as icon for this window
`InputContext getInputContext()`	It gets the input context for this window
`<T extends EventListener>T[] getListeners(Class<T> listenerType)`	It returns an array of all objects currently registered as *Foo*`Listener`s upon this window
`Locale getLocale()`	It gets the `Locale` object associated with this window, if the locale has been set
`Dialog.ModalExclusionType getModalExclusionType()`	It gets the modal exclusion type of this window
`Component getMostRecentFocusOwner()`	It gets the child component of this window that will receive the focus when this window is focused
`float getOpacity()`	It returns the opacity of the window
`Window[] getOwnedWindows()`	It returns an array containing all the windows currently owned by this window
`Window getOwner()`	It gets the owner of this window
`static Window[] getOwnerlessWindows()`	It returns an array containing all the windows created by this application that have no owner
`Shape getShape()`	It returns the shape of this window
`Toolkit getToolkit()`	It gets the frame's toolkit
`Window.Type getType()`	It returns the type of the window
`String getWarningString()`	It gets the warning string displayed with this window

Table 10.6: Methods of the Window class

Method	Does this
`WindowFocusListener[] getWindowFocusListeners()`	It returns an array of all the windows focus listeners registered on this window
`WindowListener[] getWindowListeners()`	It returns an array of all the windows listeners registered on this window
`static Window[] getWindows()`	It returns an array of all windows (both owned and ownerless) that are created by this application
`WindowStateListener[] getWindowStateListeners()`	It returns an array of all the windows state listeners registered on this window
`void hide()`	It is deprecated and replaced by `setVisible(boolean)`
`boolean isActive()`	It checks whether this window is active
`boolean isAlwaysOnTopSupported()`	It checks whether or not the always-on-top mode is supported for this window
`boolean isAutoRequestFocus()`	It checks whether this window should receive the focus on being shown subsequently or when it is moved to the front
`boolean isFocusableWindow()`	It checks whether this window or any of its child components can become the focus owner
`boolean isFocusCycleRoot()`	It always returns True as all windows are roots of a focus traversal cycle
`boolean isFocused()`	It checks whether this window is focused
`boolean isLocationByPlatform()`	It returns True if this window will appear at the default location for the native windowing system the next time this window is visible
`boolean isOpaque()`	It checks if this window is currently opaque
`boolean isShowing()`	It checks if this window is showing on the screen
`boolean isValidateRoot()`	It checks whether this container is a validate root
`void pack()`	It causes this window to be sized to fit the preferred size and layout of its sub components
`void paint(Graphics g)`	It paints the container
`boolean postEvent(Event e)`	It is deprecated and replaced by `dispatchEvent(AWTEvent)`
`protected void processEvent (AWTEvent e)`	It processes events on this window
`protected void processWindowEvent (WindowEvent e)`	It processes window events occurring on this window by sending them to any registered `WindowListener` objects
`protected void processWindowFocusEvent (WindowEvent e)`	It processes window focus event occurring on this window by sending them to any registered `WindowFocusListener` objects
`protected void processWindowStateEvent (WindowEvent e)`	It processes window state event occurring on this window by sending them to any registered `WindowStateListener` objects
`void removeNotify()`	It makes this container undisplayable by removing its connection to its native screen resource
`void removeWindowFocusListener (WindowFocusListener l)`	It removes the given window focus listener so that it no longer receives window events from this window
`void removeWindowListener (WindowListener l)`	It removes the given window listener so that it no longer receives window events from this window
`void removeWindowStateListener (WindowStateListener l)`	It removes the given window state listener so that it no longer receives window events from this window

Table 10.6: Methods of the Window class

Method	Does this
`void reshape(int x, int y, int width, int height)`	It is deprecated and replaced by `setBounds(int, int, int, int)`
`void setAlwaysOnTop(Boolean alwaysOnTop)`	It sets whether this window should always be on top of other windows or not
`void setAutoRequestFocus(boolean autoRequestFocus)`	It sets whether this window should receive the focus on being shown subsequently or when it is moved to the front
`void setBackground(Color bgColor)`	It sets the background color of this window
`void setBounds(int x, int y, int width, int height)`	It moves and resizes this component
`void setBounds(Rectangle r)`	It moves and resizes this component to conform to the new bounding rectangle
`void setCursor (Cursor cursor)`	It sets the cursor image to a given cursor
`void setFocusableWindowState (boolean focusableWindowState)`	It sets whether this window can become the focused window if it meets the other requirements outlined in the `isFocusableWindow`,
`void setFocusCycleRoot(boolean focusCycleRoot)`	It doesn't do anything as windows must always be roots of a focus traversal cycle
`void setIconImage(Image image)`	It sets the image which is displayed as icon image for this window
`void setIconImages(List<? Extends Image> icons)`	It sets the sequence of images to be displayed as icon for this window
`void setLocation(int x, int y)`	It moves the component to a new location
`void setLocation(Point p)`	It moves the component to a new location
`void setLocationByPlatform(boolean locationbyPlatform)`	It sets whether this window should appear at the default location for the native windowing system or at the current location
`void setLocationRelativeTo (Component c)`	It sets the location of this window relative to the specified component
`void setMinimumSize(Dimension minimumSize)`	It sets the minimum size of this window to a constant value
`void setModalExclusionType (Dialog.ModalExclusionType exclusionType)`	It sets the modal exclusion type for this window
`void setOpacity (float opacity)`	It sets the opacity of this window
`void setShape(Shape shape)`	It sets the shape of this window
`void setSize(Dimension d)`	It resizes this component so that it has width `d.width` and height `d.height`
`void setSize(int width, int height)`	It resizes this component so that it has width `width` and height `height`
`void setType(Window.Type type)`	It sets the type of the window
`void setVisible(boolean b)`	It sets the visibility of this window
`void show()`	It is deprecated and replaced by `setVisible(boolean`
`void toBack()`	It sends this window to the back (if this window is visible) or may cause it to lose focus or activation if it is the focused or active window
`void toFront()`	It brings this window to the front(if this window is visible) and may make it a focused window

Let's look at an example. In this case, we create a new window class named `WindowDemo` that displays a label with the text "Kogent Learning Solutions!" we start by declaring this new class:

```
class WindowDemo extends Window { . . . }
```

We have to pass the `Window` class's constructor a `Frame` window object or another `Window` object. To make things simpler, we use this new window in an application, so we'll pass the main `Frame` window object to the `WindowDemo` class's constructor and then pass that object back to the `Window` class's constructor, like this:

```
class WindowDemo extends Window {
Label label1;
WindowDemo(ShowHideWindow shw) {
super(shw);
. . . }
}
```

As is the case with frame windows, the default layout manager here is a border layout manager, so we'll set a flow layout and display the label with its message, "Kogent Learning Solutions!", in the window:

```
class WindowDemo extends Window {
Label label1;
WindowDemo(ShowHideWindow shw) {
super(shw);
setLayout(new FlowLayout());
label1 = new Label("Kogent Learning Solutions!");
add(label1); }
. . .
}
```

Windows of the `Window` class are simply blank white spaces, so we'll also override the `paint()` method of this window to add a very simple frame—just a rectangle— like this (note that we can determine the dimensions of the window with the `getSize()` method, the same as with applets):

```
class WindowDemo extends Window {
Label label1;
WindowDemo(ShowHideWindow shw) {
super(shw);
setLayout(new FlowLayout());
label1 = new Label("Kogent Learning Solutions!");
add(label1); }
public void paint(Graphics g) {
int width = getSize().width;
int height = getSize().height;
g.drawRect(0, 0, --width, --height); }
}
```

That completes the `WindowDemo` class. We'll put this class to work in an application now. In this case, we'll just instantiate an object of the class and set its size— this is required before the window can be displayed. We'll also set its screen location using `setLocation() method`, because there's currently no way for the user to move this window when it appears; and if we just show it with the `setVisible()` method, it'll appear at (0, 0) on the screen, which is at the extreme upper-left corner. When the user enters "show" in the text field, the window appears, and when the user enters "hide", the window gets disappeared. Here's the code:

```
import java.awt.*;
import java.awt.event.*;
public class ShowHideWindow extends Frame implements ActionListener{
  TextField text1;
  WindowDemo window;
  ShowHideWindow(){
        text1 = new TextField(30);
        text1.addActionListener(this);
        add(text1);
        window = new WindowDemo(this);
        window.setSize(300, 200);
        window.setLocation(300, 300);
        setSize(300,200);
```

```
            setVisible(true);
      }
      public void actionPerformed(ActionEvent e) {
         if(text1.getText().equalsIgnoreCase("show"))window.setVisible(true);
         if(text1.getText().equalsIgnoreCase("hide"))window.setVisible(false); }
      public static void main(String args[]){ new ShowHideWindow(); }

   }
   class WindowDemo extends Window {
   Label label1;
   WindowDemo(ShowHideWindow shw) {
   super(shw);
   setLayout(new FlowLayout());
   label1 = new Label("Kogent Learning Solutions!");
   add(label1); }
   public void paint (Graphics g) {
   int width = getSize().width;
   int height = getSize().height;
   g.drawRect(0, 0, --width, --height); }
   }
```

You can see the result in Figure 10.3. As you can see in this figure, the window appears with its minimal border and the label control, which displays a message.

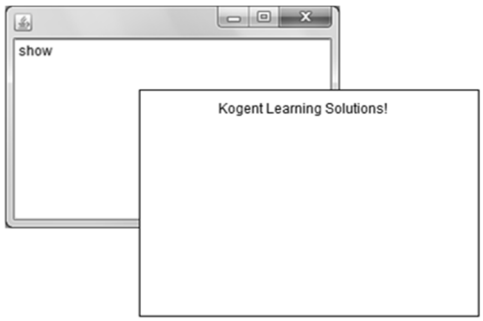


Figure 10.3: Creating a window of the Window class

Here is another example. In this case, we create a class `ListenerWindow` extended from `Frame` and will also implement `WindowListener`:

```
   import java.awt.event. *;
   class ListenerWindow extends Frame implements WindowListener { . . . }
```

To complete the implementation of the windows listener, we'll override the methods:

```
   public void windowClosing(WindowEvent event) { }
   public void windowClosed(WindowEvent event) { }
   public void windowDeiconified(WindowEvent event) { }
   public void windowIconified(WindowEvent event) { }
   public void windowActivated(WindowEvent event) { }
   public void windowDeactivated(WindowEvent event) { }
   public void windowOpened(WindowEvent event) { }
```

Similarly we can implement other listeners of window class like `windowFocusListener`:

```
   import java.awt.*;
   import java.awt.event.*;
   class ListenerWindow extends Frame implements WindowListener,
   WindowFocusListener, WindowStateListener {
   final static String line = "\n";
   TextArea text1;
   public ListenerWindow(String title) {
   super(title);
   text1 = new TextArea(5,20);
   text1.setEditable(false);
```

```
    ScrollPane scrollPane = new ScrollPane();
    scrollPane.setSize(new Dimension(300,250));
    scrollPane.add(text1);
    add(scrollPane);
    addWindowListener(this);;
    addWindowFocusListener(this);
    addWindowStateListener(this);
    setSize(400,400);
    setVisible(true); }
    public void windowClosing(WindowEvent e) {
    text1.append("WindowListener(Closing) method called." + line); }
    public void windowClosed(WindowEvent e) {
    text1.append("WindowListener(Closed) method called" + line); }
    public void windowOpened(WindowEvent e) {
    text1.append("WindowListener(Opened) method called" + line); }
    public void windowIconified(WindowEvent e) {
    text1.append("WindowListener(Iconfied) method called." + line); }
    public void windowDeiconified(WindowEvent e) {
    text1.append("WindowListener(Deiconified) method called." + line); }
    public void windowActivated(WindowEvent e) {
    text1.append("WindowListener(Activated) method called" + line); }
    public void windowDeactivated(WindowEvent e) {
    text1.append("WindowListener(Deactivated) method called" + line); }
    public void windowGainedFocus(WindowEvent e) {
    text1.append("WindowFocusListener(GainedFocus) method called" + line); }
    public void windowLostFocus(WindowEvent e) {
    text1.append("WindowFocusListener(LostFocus) method called" + line); }
    public void windowStateChanged(WindowEvent e) {
    text1.append( "WindowStateListener(StateChanged) method called"+
     line); }
    }
    public class WindowListenerDemo {
    public static void main(String[] args) {
    ListenerWindow wl=new ListenerWindow ("Windows Event Handling"); }
    }
```

The result of the code is displayed in Figure 10.4.

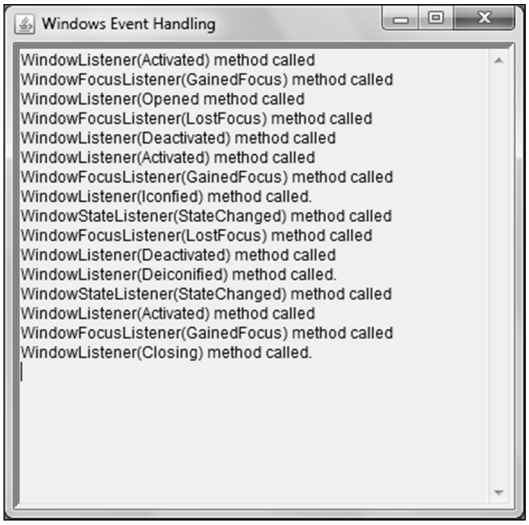


Figure 10.4: Windows Listener events

Creating Menus

"OK," says the Novice Programmer, "I have been waiting to ask you this one." "Oh boy," you say. The NP asks, "How do I create menus?" You sit back and say, "Better get some coffee."

You can add AWT menus to `Frame` class windows using three AWT classes: `MenuBar`, `Menu`, and `MenuItem`. We list the details for these classes here. The first class you use is the `MenuBar` class, which adds a menu bar to a frame window. After you have added a menu bar to a frame window, you can add menus, such as a File menu and an Edit menu, to that menu bar using the `Menu` class. Finally, you add menu items to the menus in your program using the `MenuItem` class.

We'll create and use menus over the next few solutions while developing an applet named `MenuDemo.java`,. This applet will display a frame window with a menu in it. Here's the applet:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
/* <APPLET
  CODE=MenuDemo.class
        WIDTH=300
        HEIGHT=200>
</APPLET> */
public class MenuDemo extends Applet implements ActionListener{
TextField text1;
MenuFrame frame;
public void init() {
text1 = new TextField(30);
add(text1);
text1.addActionListener(this);
frame = new MenuFrame("Menu Frame");
frame.setSize(300,200); }
  public void actionPerformed(ActionEvent event) {
        String text = text1.getText();
        if(text.equalsIgnoreCase("show")){ frame.setVisible(true); }
        else if(text.equalsIgnoreCase("hide")){ frame.setVisible(false); }
else text1.setText("Please enter either show or hide");}
}
class MenuFrame extends Frame implements ActionListener {
Menu menu1;
MenuBar menubar1;
MenuItem item1, item2, item3;
Label label1;
MenuFrame(String title) {
super(title);
label1 = new Label();
add(label1);
menubar1 = new MenuBar();
menu1 = new Menu("Food Items");
item1 = new MenuItem("Pizza");
menu1.add(item1);
item1.addActionListener(this);
item2 = new MenuItem("Burger");
menu1.add(item2);
item2.addActionListener(this);
item3 = new MenuItem("Idli");
menu1.add(item3);
item3.addActionListener(this);
menubar1.add(menu1);
setMenuBar(menubar1);
}
```

```
    public void actionPerformed(ActionEvent e) {
    if(e.getSource() == item1) {
      label1.setText("You ordered Pizza"); }
    if(e.getSource() == item2) {
      label1.setText("You ordered Burger"); }
    if(e.getSource() == item3) {
      label1.setText("You ordered Idli"); }
    }
    }
```

You can see this applet at work in Figure 10.5. The menu work in this applet takes place in the class we have named `MenuFrame`.

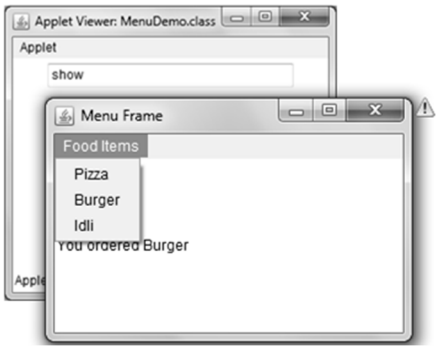


Figure 10.5: Applet showing menu system

We develop this class throughout the examples shown in the next few solutions.

Creating a **MenuBar** Object

"I'm ready to create a menu system in my frame window," says the Novice Programmer. "Where do I start?" You say, "You start by creating a `MenuBar` object. Sit down and I'll show you how."

To create a menu bar in a frame window, you use the `MenuBar` class. Here's the inheritance diagram for the `MenuBar` class:

```
    java.lang.Object
      |____java.awt.MenuComponent
            |___java.awt.MenuBar
```

You'll find the constructor for the `MenuBar` class in Table 10.7 and its methods in Table 10.8.

Table 10.7: The constructor of the MenuBar class

Constructor	Does this
`MenuBar()`	It creates a new menu bar

Table 10.8: Methods of the MenuBar class

Method	Does this
`Menu add(Menu m)`	It adds the given menu to the menu bar
`void addNotify()`	It creates the menu bar's peer
`int countMenus()`	It is deprecated and replaced by `getMenuCount()`
`void deleteShortcut (MenuShortcut s)`	It deletes the given menu shortcut
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this menu bar
`Menu getHelpMenu()`	It gets the help menu on the menu bar
`Menu getMenu(int i)`	It gets the given menu

Table 10.8: Methods of the MenuBar class	
Method	**Does this**
`int getMenuCount()`	It gets the number of menus on the menu bar
`MenuItem getShortcutMenuItem (MenuShortcut s)`	It gets the instance of `MenuItem` associated with the given `MenuShortcut` object
`void remove(int index)`	It removes the menu located at the given index from this menu bar
`void remove (MenuComponent m)`	It removes the given menu component from this menu bar
`void removeNotify()`	It removes the menu bar's peer
`void setHelpMenu(Menu m)`	It sets the help menu on this menu bar to be the given menu
`Enumeration<MenuShortcut> shortcuts()`	It gets an enumeration of all menu shortcuts managed by this menu bar

Here's how we create a menu bar in the frame window of the `MenuDemo.java` applet introduced in the "Creating Menus" solution (note that we install the menu bar in the frame window with the `Frame` class's `setMenuBar()` method):

```
class MenuFrame extends Frame implements ActionListener {
MenuBar menubar1;
Label label1;
MenuFrame(String title) {
super(title);
label1 = new Label();
add(label1);
menubar1 = new MenuBar();
. . .
setMenuBar(menubar1); }
}
```

All this does, however, is display a blank menu bar—it's time to add some menus to the menu bar. See the next solution for details.

You can find the process of Creating a Menu Bar in detail in Chapter 16.

Creating **Menu** Objects

"Well," says the Novice Programmer, "I have found that after creating a new menu bar, nothing appears in it. How do I add menus to my menu bar?" "That's easy." you say, "Just use the `Menu` class." You create the individual menus in a menu bar with the `Menu` class. Here's the inheritance diagram for this class:

```
java.lang.Object
  |____java.awt.MenuComponent
        |____java.awt.MenuItem
              |____java.awt.Menu
```

You'll find the `Menu` class's constructors in Table 10.9 and its methods in Table 10.10.

Table 10.9: Constructors of the Menu class	
Constructor	**Does this**
`Menu()`	It creates a new menu
`Menu(String label)`	It creates a new menu with the given label
`Menu(String label, boolean tearOff)`	It creates a new menu with the given label, indicating whether the menu can be torn off

Table 10.10: Methods of the Menu class	
Method	**Does this**
`MenuItem add(MenuItem mi)`	It adds the given menu item to this menu
`void add(String label)`	It adds an item with the given label to this menu

Table 10.10: Methods of the Menu class	
Method	**Does this**
`void addNotify()`	It creates the menu's peer
`void addSeparator()`	It adds a separator line or a hyphen, to the menu at the current position
`int countItems()`	It is deprecated and replaced by `getItemCount()`
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this menu
`MenuItem getItem(int index)`	It gets the item located at the given index
`int getItemCount()`	It gets the number of items in this menu
`void insert (MenuItem menuitem, int index)`	It inserts a menu item into this menu at the given position
`void insert(String label, int index)`	It inserts a menu item with the given label into this menu at the given position
`void insertSeparator(int index)`	It inserts a separator at the given position
`boolean isTearOff()`	It indicates whether this menu is a tear-off menu
`String paramString()`	It gets the parameter string representing the state of this menu
`void remove(int index)`	It removes the menu item at the given index from this menu
`void remove (MenuComponent item)`	It removes the given menu item from this menu
`void removeAll()`	It removes all items from this menu
`void removeNotify()`	It removes the menu's peer

Here's how we created a `Food Items` menu in the `MenuDemo.java` applet and added that menu to the menu bar (note that we name the menu simply by passing the name to the `Menu` class's constructor):

```
class MenuFrame extends Frame implements ActionListener {
Menu menu1;
MenuBar menubar1;
Label label1;
MenuFrame(String title) {
super(title);
label1 = new Label();
add(label1);
menubar1 = new MenuBar();
menu1 = new Menu("Food Items");
. . .
menubar1.add(menu1);
setMenuBar(menubar1); }
}
```

TIP

One powerful way of creating menus is as tear-off menus, which you do by using the third form of the Menu constructor in Table 10.9. Users can "tear off" these menus with the mouse and place them where they like.

You can find the process of Creating a Menu in detail in Chapter 16.

Creating **MenuItem** Objects

The Novice Programmer is back and says, "Hey, I have created a new menu. How would I add menu items to that menu?" "It's not so hard," you say, "you just use the `MenuItem` class."

Each menu item you add to an AWT menu is actually an object of the `MenuItem` class. Here's the inheritance diagram for this class:

```
java.lang.Object
  |____java.awt.MenuComponent
        |____java.awt.MenuItem
```

You'll find the constructors for the `MenuItem` class in Table 10.11 and its methods in Table 10.12.

Table 10.11: Constructors of the MenuItem class

Constructor	Does this
`MenuItem()`	It creates a new menu item without any label or keyboard shortcut
`MenuItem(String label)`	It creates a new menu item with the given label and no keyboard shortcut
`MenuItem(String label, MenuShortcut s)`	It creates a menu item with an associated keyboard shortcut

Table 10.12: Methods of the MenuItem class

Method	Does this
`void addActionListener (ActionListener l)`	It adds the given action listener to receive action events from this menu item
`void addNotify()`	It creates the menu item's peer
`void deleteShortcut()`	It deletes any `MenuShortcut` object associated with this menu item
`void disable()`	It is deprecated and replaced by `setEnabled(boolean)`
`protected void disableEvents(long eventsToDisable)`	It disables event delivery to this menu item for events defined by the specified event mask parameter
`void enable()`	It is deprecated and replaced by `setEnabled(boolean)`
`void enable(boolean b)`	It is deprecated and replaced by `setEnabled(boolean)`
`protected void enableEvents (long eventsToEnable)`	It enables event delivery to this menu item for events defined by the specified event mask parameter
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this menu item
`String getActionCommand()`	It gets the command name of the action event that's caused by this menu item
`ActionListener[] getActionListeners()`	It gets an array of all the action listeners registered on this menu item
`String getLabel()`	It gets the label for this menu item
`<T extends EventListener>T[] getListeners(Class<T> listenerType)`	It gets an array of all objects currently registered as *Foo*`Listeners` upon this menu item
`MenuShortcut getShortcut()`	It gets the `MenuShortcut` object associated with this menu item
`boolean isEnabled()`	It checks whether this menu item is enabled
`String paramString()`	It gets the parameter string representing the state of this menu item
`protected void processActionEvent (ActionEvent e)`	It processes action events occurring on this menu item by sending them to any registered `ActionListener` objects
`protected void processEvent(AWTEvent e)`	It processes events on this menu item
`void removeActionListener (ActionListener l)`	It removes the given action listener so that it no longer receives action events from this menu item
`void setActionCommand (String command)`	It sets the command name of the action event that's fired by this menu item
`void setEnabled(boolean b)`	It sets whether this menu item can be chosen
`void setLabel(String label)`	It sets the label for this menu item to the given label
`void setShortcut (MenuShortcut s)`	It sets the `MenuShortcut` object associated with this menu item

To add a menu item to a menu, you just create a new `MenuItem` object, pass the name of new item to the `MenuItem` constructor, and then use the `Menu` object's add() method to add the new item to a menu. Here's an example in which we added items to the `Food Items` menu in the `MenuDemo.java` applet:

```
class MenuFrame extends Frame implements ActionListener {
Menu menu1;
MenuBar menubar1;
MenuItem item1, item2, item3;
Label label1;
MenuFrame(String title) {
super(title);
label1 = new Label("Hello Java!");
add(label1);
menubar1 = new MenuBar();
menu1 = new Menu("Food Items");
item1 = new MenuItem("Pizza");
menu1.add(item1);
item2 = new MenuItem("Burger");
menu1.add(item2);
item3 = new MenuItem("Idli");
menu1.add(item3);
menubar1.add(menu1);
setMenuBar(menubar1); }
}
```

TIP

Java menu items support shortcuts, which you can create by using the third form of the MenuItem constructor in Table 10.11. Users can access menu items with shortcuts using the keyboard.

This is all fine as far as it goes, except that nothing happens when these menu items are clicked. To learn how to support menu events, see the next solution.

You can find the process of Creating a Menu Item in detail in Chapter 16.

Handling Menu Events

The Novice Programmer is back and says, "Well, I have set up my entire menu system, but there's a problem. Nothing happens when I click the items in my menus." "That's because you have to add an action listener to each item," you say, smiling. "Oh," says the NP.

You use action listeners with menu items, just as you do with buttons. Here's an example in which we enable menu event handling in the `MenuDemo.java` applet. In this case, we indicate which menu item the user selected by displaying a message in a label. First, we add an action listener to each menu item:

```
class MenuFrame extends Frame implements ActionListener {
Menu menu1;
MenuBar menubar1;
MenuItem item1, item2, item3;
Label label1;
MenuFrame(String title) {
super(title);
label1 = new Label();
add(label1);
menubar1 = new MenuBar();
menu1 = new Menu("Food Items");
item1 = new MenuItem("Pizza");
menu1.add(item1);
item1.addActionListener(this);
item2 = new MenuItem("Burger");
menu1.add(item2);
item2.addActionListener(this);
item3 = new MenuItem("Idli");
```

```
    menu1.add(item3);
    item3.addActionListener(this);
    menubar1.add(menu1);
    setMenuBar(menubar1);
        }
    public void actionPerformed(ActionEvent e) {
    if(e.getSource() == item1) {
      label1.setText("You ordered Pizza"); }
    else if(e.getSource() == item2) {
      label1.setText("You selected Burger"); }
    else if(e.getSource() == item3) {
      label1.setText("You selected Idli"); }
    }
    }
```

That's it. Now the user can run the `MenuDemo.java` applet and report on the items he/she selected, as shown in Figure 10.5. It's worth noting that although we used the `ActionEvent` class's `getSource()` method to determine which menu item was clicked in this solution, you can also give each menu item an action command with the `setActionCommand()` method— as we did for buttons in Chapter 7—and use the `ActionEvent` class's `getActionCommand()` method to read the action command.

More Menu Options

In the previous few solutions, we created an applet, `MenuDemo.java`, that examined the basic construction of a menu system. In the following few solutions, we'll elaborate that menu system to include menu separators, disabled menu items, checkbox menu items, and submenus—all in a new applet called `AdvancedMenu.java`. Here's is the code for `AdvancedMenu.java`:

```
    import java.applet.Applet;
    import java.awt.*;
    import java.awt.event.*;
    /* <APPLET
            CODE=AdvancedMenu.class
            WIDTH=300
            HEIGHT=200 >
    </APPLET> */
    public class AdvancedMenu extends Applet implements ActionListener {
    TextField text1;
    MenuFrame frame;
    public void init() {
    text1 = new TextField(30);
    add(text1);
    text1.addActionListener(this);
    frame = new MenuFrame("Menus");
    frame.setSize(300, 200); }
         public void actionPerformed(ActionEvent event) {
      String text = text1.getText();
      if(text.equalsIgnoreCase("show")){ frame.setVisible(true); }
      else if(text.equalsIgnoreCase("hide")){ frame.setVisible(false); }
            else text1.setText("Please enter either show or hide");}
    }
    class MenuFrame extends Frame implements ActionListener, ItemListener {
    Menu menu1, submenu1;
    MenuBar menubar1;
    Label label1;
    MenuItem item1, item2, item4;
    MenuItem subitem1, subitem2, subitem3;
    CheckboxMenuItem item3;
    MenuFrame(String title) {
    super(title);
    label1 = new Label();
```

```
    add(label1);
    menubar1 = new MenuBar();
    menu1 = new Menu("Food Items");
    item1 = new MenuItem("Pizza");
    item1.addActionListener(this);
    menu1.add(item1);
    item2 = new MenuItem("Burger");
    item2.addActionListener(this);
    menu1.add(item2);
    menu1.addSeparator();
    item3 = new CheckboxMenuItem("Idli");
    item3.addItemListener(this);
    menu1.add(item3);
    menu1.addSeparator();
    submenu1 = new Menu("Cold Drinks");
    subitem1 = new MenuItem("Pepsi");
    subitem2 = new MenuItem("Limca");
    subitem3 = new MenuItem("Coke");
    subitem1.addActionListener(this);
    subitem2.addActionListener(this);
    subitem3.addActionListener(this);
    item2.addActionListener(this);
    item2.addActionListener(this);
    submenu1.add(subitem1);
    submenu1.add(subitem2);
    submenu1.add(subitem3);
    menu1.add(submenu1);
    menu1.addSeparator();
    item4 = new MenuItem("Exit");
    item4.addActionListener(this);
    menu1.add(item4);
    menubar1.add(menu1);
    setMenuBar(menubar1);
    }
    public void actionPerformed(ActionEvent e) {
    if(e.getSource() == item1) {
      label1.setText("You ordered Pizza"); }
    else if(e.getSource() == item2) {
      item2.setEnabled(false);
      label1.setText("You ordered Burger"); }
    else if(e.getSource() == subitem1) {
      label1.setText("You ordered Pepsi"); }
    else if(e.getSource() == subitem2) {
      label1.setText("You ordered Limca"); }
    else if(e.getSource() == subitem3) {
      label1.setText("You ordered Coke"); }
    else if(e.getSource() == item4) {
      setVisible(false); }
    }
    public void itemStateChanged (ItemEvent e) {
    if(e.getSource() == item3) {
          if(((CheckboxMenuItem)e.getItemSelectable()).getState())
    label1.setText("You ordered Idli");
          else { label1.setText("Idli not ordered"); }
             }
       }
    }
```

You can see this applet in Figure 10.6. We'll develop the various new aspects of this applet, `AdvancedMenu.java`, over the next few solutions, as you'll see how to handle its various parts in code.

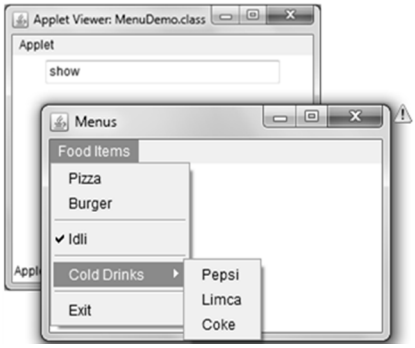


Figure 10.6: Applet showing a menu system

Adding Menu Separators

You can combine menu items into groups using *menu separators*—the thin horizontal lines that appear in menus (refer back to Figure 10.6). Adding a menu separator to a menu is easy—you just use the `addSeparator()` method, like this:

```
    MenuFrame(String title) {
    super(title);
    label1 = new Label();
    add(label1);
    menubar1 = new MenuBar();
    menu1 = new Menu("Food Items");
    item1 = new MenuItem("Pizza");
    item1.addActionListener(this);
    menu1.add(item1);
    item2 = new MenuItem("Burger");
    item2.addActionListener(this);
    menu1.add(item2);
    menu1.addSeparator();
    . . .
    }
```

Using `addSeparator()` produces the menu separators you see in Figure 10.6.

Disabling Menu Items

"Ugh that Johnson," the Novice Programmer says, "has been messing around with my program again by selecting menu items that don't apply, such as spell checking when the program was drawing graphics. It crashed my program right in front of the Big Boss!" "Hmm," you say, "how about disabling menu items when they are not appropriate?" The NP smiles and says, "Now, I'll show that to silly Johnson!"

When you disable a menu item, the menu item is displayed in gray and cannot be selected or clicked. To disable a menu item, you use that item's `setEnabled()` method, passing the method a value of False. For example, we can disable the second menu item in `AdvancedMenu.java` when the user clicks it, like this:

```
    class MenuFrame extends Frame implements ActionListener {
    Menu menu1, submenu1;
    MenuBar menubar1;
    Label label1;
    MenuItem item1, item2, item4;
    MenuItem subitem1, subitem2, subitem3;
    CheckboxMenuItem item3;
    MenuFrame(String title){
    super(title);
    label1 = new Label();
```

```
    add(label1);
    menubar1 = new MenuBar();
    menu1 = new Menu("Food Items");
    item1 = new MenuItem("Pizza");
    item1.addActionListener(this);
    menu1.add(item1);
    item2 = new MenuItem("Burger");
    item2.addActionListener(this);
    menu1.add(item2);
    menubar1.add(menu1);
    setMenuBar(menubar1); }
    public void actionPerformed(ActionEvent e) {
    if(e.getSource() == item1) {
      label1.setText("You ordered Pizza"); }
    else if(e.getSource() == item2) {
      item2.setEnabled(false);
      label1.setText("You ordered Burger"); }
    }
    }
```

The result appears in Figure 10.7, where you can see the disabled menu item:

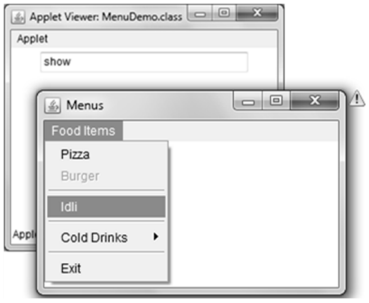


Figure 10.7: Showing a disabled menu item

Now that the item is disabled, the user has no way to click it again to enable it.

Adding Checkboxes to Menus

The Product Support Specialist is back and unhappy. "There's a problem in your new application," the PSS says, "because the user has no way of knowing whether the Auto-translate to German menu option is on or off, and the results can prove to be pretty startling." "Hmm," you say. "I suppose I can add a checkbox to that menu option to show when it's in effect."

You can create menu items that display checkmarks next to them to indicate when specific options are in effect. Also, you can toggle a checkmark when the user selects the corresponding menu item. To support checkmarks in menu items, use the `CheckboxMenuItem` class, which has the following inheritance diagram:

```
    java.lang.Object
      |____java.awt.MenuComponent
            |____java.awt.CheckboxMenuItem
```

You'll find the constructors for `CheckboxMenuItem` in Table 10.13 and its methods in Table 10.14.

Table 10.13: Constructors of the CheckboxMenuItem class

Constructor	Does this
`CheckboxMenuItem()`	It creates a checkbox menu item without any label
`CheckboxMenuItem(String label)`	It creates a checkbox menu item with the given label
`CheckboxMenuItem(String label, boolean state)`	It creates a checkbox menu item with the given label and state

Table 10.14: Methods of the CheckboxMenuItem class	
Method	**Does this**
`void addItemListener (ItemListener l)`	It adds the given item listener to receive item events from this checkbox menu item
`void addNotify()`	It creates the peer of the checkbox item
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this `CheckboxMenuItem`
`ItemListener[] getItemListeners()`	It returns an array of all the item listeners on this checkbox menu item
`<T extends EventListener>T[] getListeners (Class<T> listenerType)`	It returns an array of all the listeners currently registered as *`Foo`*`Listeners` upon this checkbox menu item
`Object[] getSelectedObjects()`	It gets an array (of length 1) containing the checkbox menu item label
`boolean getState()`	It determines whether the state of this checkbox menu item is on or off
`String paramString()`	It gets a string representing the state of this checkbox menu item
`protected void processEvent (AWTEvent e)`	It processes events on this checkbox menu item
`protected void processItemEvent (ItemEvent e)`	It processes item events occurring in this checkbox menu item by sending them to any `ItemListener` objects
`void removeItemListener (ItemListener l)`	It removes the given item listener so that it no longer receives item events from this checkbox menu item
`void setState(boolean b)`	It sets this checkbox menu item to the specified state

Note that you use item listeners, not action listeners, with checkbox menu items. Here's how we add a checkmark item to the applet `AdvnancedMenu.java` using an `ItemListener` object:

```
class MenuFrame extends Frame implements ActionListener, ItemListener {
Menu menu1, submenu1;
MenuBar menubar1;
Label label1;
MenuItem item1, item2, item4;
MenuItem subitem1, subitem2, subitem3;
CheckboxMenuItem item3;
MenuFrame(String title) {
super(title);
label1 = new Label();
add(label1);
menubar1 = new MenuBar();
menu1 = new Menu("Food Items");
item1 = new MenuItem("Pizza");
item1.addActionListener(this);
menu1.add(item1);
item2 = new MenuItem("Burger");
item2.addActionListener(this);
menu1.add(item2);
menu1.addSeparator();
item3 = new CheckboxMenuItem("Idli");
item3.addItemListener(this);
menu1.add(item3);
. . .
menubar1.add(menu1);
setMenuBar(menubar1);
}
```

As the user selects this item time after time, Java will toggle the checkmark on and off automatically. To handle events from this item, you have to override the `itemStateChanged()` method. In this case, we determine the state of the menu item and display that state in a label, like this:

```
public void itemStateChanged (ItemEvent e) {
if(e.getSource() == item3) {
```

```
    if(((CheckboxMenuItem)e.getItemSelectable()).getState())
          label1.setText("You ordered Idli");
        else { label1.setText("Idli not ordered"); }
    }
}
```

You can see the checkbox menu item in `AdvancedMenu.java` at work in Figure 10.8.

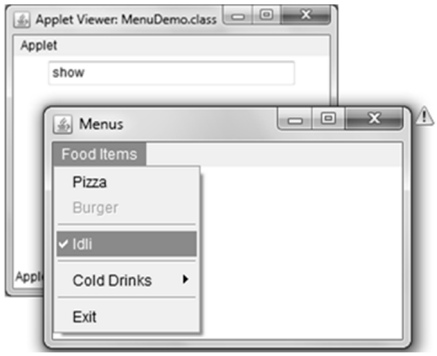


Figure 10.8: Showing a checkbox menu item

Creating Submenus

Another powerful aspect of working with menus in AWT programming involves *submenus*. Submenus are attached to a menu item; and when the user selects the menu item, the submenu opens, as you saw in Figure 10.6. The user can select items from the submenu just as he/she can select normal menus.

Using submenus is a very powerful technique when you need another level of detail; for example, you may want to let the user select a drawing color with a menu item; and when that item is selected, a submenu opens indicating possible colors, such as red, green, magenta, and blue.

Creating a submenu is easy—you just add menu items to another menu item using the `add()` method. Here's how you do it:

```
MenuFrame(String title) {
super(title);
label1 = new Label();
add(label1);
menubar1 = new MenuBar();
menu1 = new Menu("Food Items");
item1 = new MenuItem("Pizza");
item1.addActionListener(this);
menu1.add(item1);
item2 = new MenuItem("Burger");
item2.addActionListener(this);
menu1.add(item2);
menu1.addSeparator();
item3 = new CheckboxMenuItem("Idli");
item3.addItemListener(this);
menu1.add(item3);
menu1.addSeparator();
submenu1 = new Menu("Cold Drinks");
subitem1 = new MenuItem("Pepsi");
subitem2 = new MenuItem("Limca");
subitem3 = new MenuItem("Coke");
subitem1.addActionListener(this);
subitem2.addActionListener(this);
subitem3.addActionListener(this);
item2.addActionListener(this);
```

```
    item2.addActionListener(this);
    submenu1.add(subitem1);
    submenu1.add(subitem2);
    submenu1.add(subitem3);
    menu1.add(submenu1);
    menu1.addSeparator();
    item4 = new MenuItem("Exit");
    item4.addActionListener(this);
    menu1.add(item4);
    menubar1.add(menu1);
    setMenuBar(menubar1); }
```

Here's how we handle submenu item clicks in the `actionPerformed()` method:

```
    public void actionPerformed(ActionEvent e) {
    if(e.getSource() == item1) {
      label1.setText("You ordered Pizza");
    } else if(e.getSource() == item2) {
      item2.setEnabled(false);
      label1.setText("You ordered Burger");
    } else if(e.getSource() == subitem1)  {
      label1.setText("You ordered Pepsi");
    } else if(e.getSource() == subitem2) {
      label1.setText("You ordered Limca");
    } else if(e.getSource() == subitem3) {
      label1.setText("You ordered Coke");
    } else if(e.getSource() == item4) {
      setVisible(false); }
    }
```

You can see the submenu in Figure 10.9. When the user selects one of the submenu items, the applet displays which item was ordered in the label in the frame window.

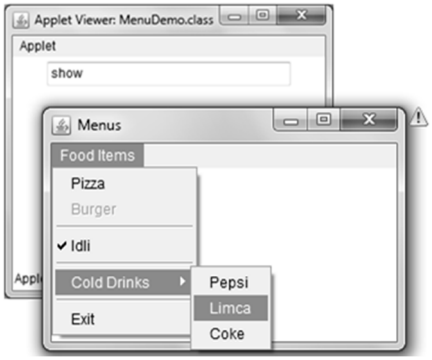


Figure 10.9: Showing submenu items

Pop-Up Menus

"Hey," says the Novice Programmer, "I saw that you can right-click some applications and a pop-up menu appears. Can I do that in Java?" "Sure thing," you say, "nothing to it. Just use the `PopupMenu` class."

You use the `PopupMenu` class to create pop-up menus that don't need to be attached to a menu bar. Here's the inheritance diagram for this class:

```
java.lang.Object
  |____java.awt.MenuComponent
        |____java.awt.MenuItem
              |____java.awt.Menu
                    |____java.awt.PopupMenu
```

You can find the constructors for the `PopupMenu` class in Table 10.15 and the methods of this class in Table 10.16.

Table 10.15: Constructors of the PopupMenu class

Constructor	Does this
`PopupMenu()`	It creates a new pop-up menu without any name
`PopupMenu(String label)`	It creates a new pop-up menu with the given name

Table 10.16: Methods of the PopupMenu class

Method	Does this
`void addNotify()`	It creates the pop-up menu's peer
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this pop-up menu
`MenuContainer getParent()`	It returns the parent container for this menu component
`void show(Component origin, int x, int y)`	It shows the pop-up menu at the x, y position relative to the component at origin

Let's look at an example. Here, we create a new pop-up menu and add four items to it (note that we also implement a `MouseListener` interface to handle right clicks):

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
/*<APPLET
         CODE=PopupDemo.class
         WIDTH=300
         HEIGHT=200 >
</APPLET>*/
public class PopupDemo extends Applet implements ActionListener,
MouseListener {
PopupMenu popup1;
MenuItem item1, item2, item3;
public void init() {
popup1 = new PopupMenu("Popup Menu");
item1 = new MenuItem("Red");
item1.addActionListener(this);
item2 = new MenuItem("Green");
item2.addActionListener(this);
item3 = new MenuItem("Blue");
item3.addActionListener(this);
popup1.add(item1);
popup1.addSeparator();
popup1.add(item2);
popup1.addSeparator();
popup1.add(item3);
add(popup1);
addMouseListener(this);
}
....
}
```

When the user right-clicks the applet window, we can use the `PopupMenu` class's `show()` method to display the menu, passing that method a `this` keyword pointing to the applet and the position at which to display the pop-up menu (here, we display the pop-up menu at the location where the mouse was clicked):

```
public void mousePressed(MouseEvent e) {
if(e.getModifiers() != 0) {
popup1.show(this, e.getX(), e.getY()); } }
```

```
            public void mouseExited(MouseEvent e){ }
            public void mouseEntered(MouseEvent e){ }
            public void mouseReleased(MouseEvent e){ }
            public void mouseClicked(MouseEvent e){ }
```

When the user clicks a submenu item, we can display which item was clicked in the `actionPerformed()` method:

```
public void actionPerformed(ActionEvent event) {
if(event.getSource() == item1)
  setBackground(Color.red);
else if(event.getSource() == item2)
  setBackground(Color.green);
  else if(event.getSource() == item3)
  setBackground(Color.blue);
}
```

The result appears in Figure 10.10. As you can see in this figure, the user can open the pop-up menu just by right clicking the applet. When the user selects a color item from the pop-up menu, the applet's background color will change accordingly.

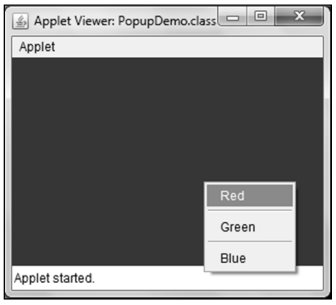


Figure 10.10: Using pop-up menu

Dialog Boxes

"Uh-oh," says the Novice Programmer, "I need some text input from the user, but the Big Boss said that if I put one more text field into my program, I would live to regret it." "How many text fields do you have in your program?" you ask. "About four hundred," the Novice Programmer says. "Hmm," you say, "well, you can always use a dialog box to get input from the user if you don't want to use another text field." The NP says, "Great!"

AWT supports a special window class—the `Dialog` class—that you can use to create dialog boxes. The windows created with this class look more like standard dialog boxes than the one you create with frame window—for example, dialog boxes do not have minimize and maximize buttons. Here's the inheritance diagram for the `Dialog` class:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
                |____java.awt.Window
                        |____java.awt.Dialog
```

You'll find the constructors of the `Dialog` class in Table 10.17 and its methods in Table 10.18.

Table 10.17: Constructors of the Dialog class

Constructor	Does this
`Dialog(Dialog owner)`	It creates an initially invisible, modeless `Dialog` object with an empty title and the given owner `Dialog`
`Dialog(Dialog owner, String title)`	It creates an initially invisible, modeless `Dialog` object with the given owner `Dialog` and title

Table 10.17: Constructors of the Dialog class	
Constructor	**Does this**
`Dialog(Dialog owner, String title, boolean modal)`	It creates an initially invisible `Dialog` object with the given owner `Dialog`, title, and modality
`Dialog(Dialog owner, String title, boolean modal GraphicsConfiguration gc)`	It creates an initially invisible `Dialog` object with the given owner `Dialog`, title, modality and graphics configuration
`Dialog(Frame owner)`	It creates an initially invisible, modeless `Dialog` object with an empty title and the given owner frame
`Dialog(Frame owner, boolean modal)`	It creates an initially invisible `Dialog` object with an empty title as well as the given owner frame and modality
`Dialog(Frame owner, String title)`	It creates an initially invisible, modeless `Dialog` object with the given owner frame and title
`Dialog(Frame owner, String title, boolean modal)`	It creates an initially invisible `Dialog` object with the given owner frame, title, and modality
`Dialog(Frame owner, String title, boolean modal, GraphicsConfiguration gc)`	It creates an initially invisible `Dialog` object with the given owner frame, title, modality, and graphics configuration
`Dialog(Window owner)`	It creates an initially invisible, modeless `Dialog` object with the given owner `Window`, and title
`Dialog(Window owner, Dialog.ModalityType modalityType)`	It creates an initially invisible, modeless `Dialog` object with the given owner `Window` and modality
`Dialog(Window owner, String title)`	It creates an initially invisible, modeless `Dialog` object with the given owner `Window` and title
`Dialog(Window owner, String title, Dialog.ModalityType modalityType)`	It creates an initially invisible, modeless `Dialog` object with the given owner `Window`, title, and modality
`Dialog(Window owner, String title, Dialog.ModalityType modalityType, GraphicsConfiguration gc)`	It creates an initially invisible, modeless `Dialog` object with the given owner `Window`, title, modality and graphics configuration

Table 10.18: Methods of the Dialog class	
Method	**Does this**
`void addNotify()`	It makes this dialog box displayable by connecting it to a native screen resource
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this dialog box
`Dialog.ModalityType getModalityType()`	It returns the modality type of this dialog box
`String getTitle()`	It gets the title of the dialog box
`void hide()`	It is deprecated and replaced by `setVisible(boolean)`
`boolean isModal()`	It indicates whether the dialog box is modal
`boolean isResizable()`	It indicates whether this dialog box can be resized by the user
`boolean isUndecorated()`	It indicates whether this dialog box is undecorated
`protected String paramString()`	It gets the string representing the state of this dialog box
`void setModal(boolean modal)`	It specifies whether this dialog box should be modal
`void setModalityType (Dialog.ModalityType type)`	It sets the modality type for this dialog box

Table 10.18: Methods of the Dialog class	
Method	**Does this**
`void setOpacity (float opacity)`	It sets the opacity of the dialog box
`void setResizable (boolean resizable)`	It sets whether this dialog box can be resized by the user
`void setShape(Shape shape)`	It sets the shape of the dialog box
`void setTitle(String title)`	It sets the title of the dialog box
`void setUndecorated (boolean undecorated)`	It disables or enables decorations for this dialog
`void setVisible (boolean b)`	It shows or hides the dialog box depending on the value of parameter `b`
`void show()`	It is deprecated and replaced by `setVisible(Boolean)`
`void toBack()`	It sends the window to the back if this window is visible and may cause it to lose focus or activation in case it is the focused or active window

There are a few things to note in Tables 11.17 and 11.18. One is that you can create either modal dialog boxes (that is, the user must dismiss the dialog box from the screen before interacting with the rest of the program) or nonmodal dialog boxes, depending on which `Dialog` constructor you use or if you use the `setModal()` method. Another point is that you can use the `setResizeable()` method to create a dialog box that the user can resize.

Let's look at an example that puts the `Dialog` class to work. We'll start by creating a new dialog box class named `OkCancelDialog` that displays a text field and two buttons: OK and Cancel. If the user types text into the text field and clicks the OK button, that text will appear in the main window of the application; otherwise, if the user clicks the Cancel button, no text will appear. We start by extending `OkCancelDialog` from the `Dialog` class, creating a new dialog box, and adding the controls we'll need in it:

```
class OkCancelDialog extends Dialog implements ActionListener {
Button ok, cancel;
TextField text;
public String data;
OkCancelDialog(Frame hostFrame, String title, boolean dModal) {
super(hostFrame, title, dModal);
setSize(300,100);
setLayout(new FlowLayout());
ok = new Button("OK"); add(ok);
ok.addActionListener((ActionListener)this);
cancel = new Button("Cancel"); add(cancel);
cancel.addActionListener(this);
text = new TextField(30); add(text);
data = new String(""); }
. . .
}
```

When the user clicks the OK button, we store the text the user typed in a public data member named `data` so that it'll be accessible to the rest of the program in the `actionPerformed()` method. If the user clicks Cancel, we place an empty string in that data member. When the user clicks either of the buttons, we also hide the dialog box. Here's the code:

```
public void actionPerformed(ActionEvent e) {
  if(e.getSource() == ok) { data = text.getText(); }
  else { data = ""; }
  setVisible(false);
. . .
}
```

That's it. We can make use of the new `OkCancelDialog` class in an application, displaying a dialog box of that class when the user selects the item, "Dialog box", in the File menu:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
public class DialogBoxDemo {
public static void main(String [] args) {
DialogFrame frame = new DialogFrame("Dialogs");
frame.setSize(200,200);
frame.addWindowListener(new WindowAdapter() { public void
windowClosing(WindowEvent e) {System.exit(0); } });
frame.setVisible(true); }
}
class DialogFrame extends Frame implements ActionListener {
Menu menu1;
MenuBar menubar1;
MenuItem item1;
Label label1;
OkCancelDialog dialog;
DialogFrame(String title) {
super(title);
label1 = new Label();
add(label1);
menubar1 = new MenuBar();
menu1 = new Menu("File");
item1 = new MenuItem("Dialog box");
menu1.add(item1);
item1.addActionListener(this);
menubar1.add(menu1);
setMenuBar(menubar1);
dialog = new OkCancelDialog(this, "Dialog", true); }
public void actionPerformed(ActionEvent e) {
if(e.getSource() == item1) {
   dialog.setVisible(true);
   label1.setText(dialog.data); }
}
}
```

Note that we also recover the text data from the dialog box using a public data member named `data` and place it in a label in the application. When this application runs and the Dialog box item in the File menu is clicked, the dialog box appears, as shown in Figure 10.11.



Figure 10.11: Displaying a dialog box

When the user enters text into the text field in the dialog box and clicks OK, the dialog box closes and that text appears in the label in the main application window, as shown in Figure 10.12. There's another kind of dialog box that's worth taking a look at—the file dialog box. We'll take a look at it next.

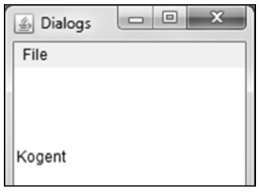


Figure 10.12: Displaying text from a dialog box

File Dialog Boxes

You can use the special AWT class `FileDialog` to create and display file dialog boxes that let the user browse directories and select files. Here's the inheritance diagram for the `FileDialog` class:

```
java.lang.Object
   |____java.awt.Component
         |____java.awt.Container
                |____java.awt.Window
                       |____java.awt.Dialog
                              |____java.awt.FileDialog
```

You'll find the fields of the `FileDialog` class in Table 10.19, its constructors in Table 10.20, and its methods in Table 10.21.

Table 10.19: Fields of the FileDialog class

Field	Does this
`static int LOAD`	It indicates that the purpose of the file dialog box is to locate a file from which to read
`static int SAVE`	It indicates that the purpose of the file dialog box is to locate a file to which to write

Table 10.20: Constructors of the FileDialog class

Constructor	Does this
`FileDialog(Dialog parent)`	It creates a file dialog for loading a file
`FileDialog(Dialog parent, String title)`	It creates a file dialog window with the given title for loading a file
`FileDialog(Dialog parent, String title, int mode)`	It creates a file dialog window with the given title for loading or saving a file
`FileDialog(Frame parent)`	It creates a file dialog box for loading a file
`FileDialog(Frame parent, String title)`	It creates a file dialog box with the given title for loading a file
`FileDialog(Frame parent, String title, int mode)`	It creates a file dialog box with the given title for loading or saving a file

Table 10.21: Methods of the FileDialog class

Method	Does this
`void addNotify()`	It creates the file dialog box's peer
`String getDirectory()`	It gets the directory of this file dialog box
`String getFile()`	It gets the selected file of this file dialog box
`FilenameFilter getFilenameFilter()`	It determines this file dialog box's file name filter
`File[] getFiles()`	It returns files selected by the user
`int getMode()`	It indicates whether this file dialog box is for loading from a file or for saving to a file

Table 10.21: Methods of the FileDialog class	
Method	**Does this**
`boolean isMultipleMode()`	It indicates whether this file dialog box allows multiple file selection
`protected String paramString()`	It gets the string representing the state of this file dialog box
`void setDirectory (String dir)`	It sets the directory of this file dialog box to the given directory
`void setFile(String file)`	It sets the selected file for this file dialog box to the given file
`void setFilenameFilter (FilenameFilter filter)`	It sets the file name filter for this file dialog box to the given filter
`void setMode(int mode)`	It sets the mode of the file dialog box
`void setMultipleMode (boolean enable)`	It enables or disables multiple file selection for the file dialog box

TIP

You can create file dialog boxes that indicate to the user that they are specifically intended for loading or saving files, depending on which constant from Table 10.19 you use with the FileDialog constructor. If you don't specify which purpose the file dialog box is for, you'll just get a generic version.

Here's an example in which we let the user browse through directories with a file dialog box. When the user selects a file, we display the file name in the main application window. Doing this is easy; we just create a `FileDialog` object, show it with the `setVisible()` method, and then read the name of the file the user selected by using the `getFile()` method:

```
import java.awt.*;
import java.awt.event.*;
public class FileDialogBoxDemo {
public static void main(String [] args) {
Dialogframe frame = new Dialogframe("Dialogs");
frame.setSize(200, 200);
frame.addWindowListener(new WindowAdapter() { public void
windowClosing(WindowEvent e) {System.exit(0); } });
frame.setVisible(true); }
}
class Dialogframe extends Frame implements ActionListener {
Menu menu1;
MenuBar menubar1;
MenuItem item1, item2;
Label label1;
FileDialog file, upload;
Dialog dialog;
Dialogframe(String title) {
super(title);
dialog = new Dialog(this, "Upload File", false, null);
label1 = new Label();
add(label1);
menubar1 = new MenuBar();
menu1 = new Menu("File");
item1 = new MenuItem("Open file");
item2 = new MenuItem("Upload file");
menu1.add(item1);
menu1.add(item2);
item1.addActionListener(this);
item2.addActionListener(this);
menubar1.add(menu1);
setMenuBar(menubar1);
file = new FileDialog(this, "File Dialog");
upload = new FileDialog(dialog, "Upload File"); }
public void actionPerformed(ActionEvent e) {
if(e.getSource() == item1) {
```

```
    file.setVisible(true);
    label1.setText("You selected " + file.getFile()); }
    if(e.getSource() == item2) { upload.setVisible(true); }
    }
    }
```

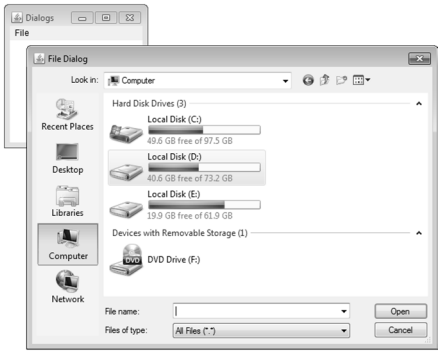


Figure 10.13: Showing a File Dialog box



Figure 10.14: Reading a file name from a File Dialog box

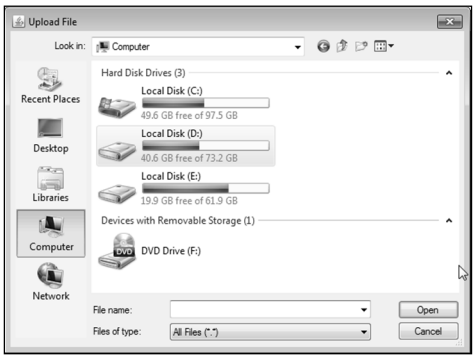


Figure 10.15: Showing an Upload File dialog box

Figure 10.13 shows the result in Windows (so a Windows File Dialog box is used). In this figure, we are navigating to a file. In Figure 10.14, we have selected a file and closed the dialog box, and the main application is displaying the file we selected. Figure 10.15 shows the Upload File dialog box.

Summary

In this chapter, you came to know about the windows, menus, and dialog boxes. Then, you learned how to create frame windows, show and hide windows, automatically hide windows upon closing, and use the `Windows` class. The chapter next helped you to create menus with menu objects and menu events, and add menu separators to them. You also learned how to create submenus and pop-up menus. Lastly, you were familiarized with the creation of dialog boxes.

In the next chapter, you will learn about Swing- Applets, Applications, and Pluggable Look and Feel.



11

Swing—Applets, Applications, and Pluggable Look and Feel

If you need an immediate solution to:	*See page:*
Working with Swing	358
Preparing to Create a Swing Applet	364
Understanding Content Panes	370
Creating a Swing Applet	372
Creating a Swing Application	374
Closing JFrame Windows	377
Using Insets for Selecting Component Borders	379
The Synth Skinnable Look and Feel	383
The Pluggable Look and Feel	386
The Pluggable Look and Feel for Components	390
The Nimbus Look and Feel	392

In Depth

This chapter begins with a discussion that a lot of programmers have been waiting for—a discussion on Swing. Let's get right into the solution with a look at some Java history.

The Java Foundation Classes

The introduction of Abstract Window Toolkit (AWT) has raised the popularity of Java with powerful Graphical User Interface (GUI) features. However, in spite of providing significant advancement, it was unable to meet the requirements of programmers. Now, it's the turn of Swing to further raise the standards and popularity of Java. The Swing components are much improved alternates compared to the AWT components and meant to fulfill serious UI needs by eliminating the limitations and bugs observed in AWT components. However, they still depend on the fundamental classes in the `java.awt` package, so AWT cannot be totally ignored.

It has been aforementioned in this book that the original AWT (took only six weeks to write) was modeled after HyperText Markup Language (HTML) controls, and one operating system window was allocated per component. The programmers, however, are accustomed to using a lot more controls; therefore, several third parties began creating their own sets of controls for use with Java. For example, Netscape introduced the popular Internet Foundation Classes (IFC) library, and the popularity of this library made Sun in deciding to create Swing components as a part of Java Foundation Classes (JFC). This was a consequence of the joint effort between the Sun and Netscape developers. Most programmers consider that Swing and JFC are same, but it is not correct. Swing along with several other items is contained in JFC. The JFC is made up of the following features:

- **UI Components**—The JFC includes everything from buttons to split panes to tables. A few of the supported features to be mentioned are many components that are capable of sorting, printing, and drag and drop.
- **Pluggable Look-and-Feel Support**—The look and feel of Swing applications is pluggable, allowing various selections of the look and feel.
- **Accessibility Application Program Interface (API)**—This feature enables assistive technologies, such as screen readers and Braille displays, to get information from the UI.
- **Java 2D API**—This feature enables developers to easily incorporate high-quality 2D graphics, text, and images in applications and applets. Java 2D includes extensive APIs for generating and sending high-quality output to printing devices.
- **Internationalization**—This feature allows developers to build applications that can interact with users all over the globe in their own languages and cultural conventions. With the input method framework, developers can build applications that accept text in languages that use thousands of different characters, such as Japanese, Chinese, or Korean.

Swing came up with three significant advancements, that is, less consumption of system resources, a lot more sophisticated components, as shown in Figure 11.1, and customizable look and feel for programs.

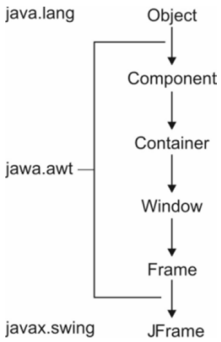


Figure 11.1: Components of java.lang, java.awt, and javax.swing

The next section discusses in detail about Swing.

Swing

The Swing packages are built on a large number of built-in AWT classes and components (over 250 classes and 40 UI components). Generally, the UI components entice the programmers, so we begin our discussion with them. Note that each UI component begins with the letter `J`; therefore, many programmers mistakenly use the terms *JFC* and *Swing* interchangeably. The different Swing components classes are enlisted as follows:

- `JApplet`—It is an extended version of `java.applet.Applet` that adds support for root panes and other panes
- `JButton`—It is a push or command button
- `JCheckBox`—It is a checkbox that can be selected or deselected and displays its state visually
- `JCheckBoxMenuItem`—It is a menu item that can be selected or deselected and displays its state visually
- `JColorChooser`—It is a pane of controls for allowesing a user to select a color
- `JComboBox`—It is a combo box, which is a combination of a text field and a drop-down list
- `JComponent`—It is the base class for the Swing components
- `JDesktopPane`—It is a container utilized for creating a multidocument interface or a desktop
- `JDialog`—It is the base class for creating a dialog window
- `JEditorPane`—It is a text component that allows the user to edit various kinds of content
- `JFileChooser`—It lets the user choose a file
- `JFrame`—It is an extended version of `java.awt.Frame` that adds support root panes and other panes
- `JInternalFrame`—It is a lightweight object that provides many of the features of a heavyweight frame
- `JInternalFrame.JDesktopIcon`—It represents an iconified version of a `JInternalFrame` component
- `JLabel`—It is a display area for a short text string or an image (or both)
- `JLayeredPane`—It adds layers to a Swing container, allowing components to overlap each other
- `JList`—It is a component that allows the user to select one or more objects from a list
- `JMenu`—It is a pop-up menu containing `JMenuItem` objects that are displayed when the user selects it in the `JMenuBar` component
- `JMenuBar`—It is an implementation of a menu bar
- `JMenuItem`—It is an implementation of a menu item
- `JOptionPane`—It makes it easy to pop up a standard dialog box
- `JPanel`—It is a generic lightweight container
- `JPasswordField`—It allows editing of a single line of text, where the view does not show the original characters
- `JPopupMenu`—It is a pop-up menu
- `JPopupMenu.Separator`—It is a pop-up menu-specific separator
- `JProgressBar`—It is a component that displays an integer value within an interval
- `JRadioButton`—It is a radio button that can be selected or deselected and displays its state visually
- `JRadioButtonMenuItem`—It is a radio button menu item
- `JRootPane`—It is the fundamental component in the container hierarchy
- `JScrollBar`—It is an implementation of a scrollbar
- `JScrollPane`—It refers to a container that handles a viewport, optional vertical and horizontal scrollbars, as well as optional row and column heading viewports
- `JSeparator`—It is a menu separator
- `JSlider`—It is a component that lets the user select a value by sliding a knob within an interval

- `JSplitPane`—It divides two components
- `JTabbedPane`—It lets the user switch between groups of components by clicking tabs
- `JTable`—It presents data in a two-dimensional table format
- `JTextArea`—It is a multiline area that displays plain text
- `JTextField`—It allows the editing of a single line of text
- `JTextPane`—It is a text component that can be marked up with attributes
- `JToggleButton`—It is a two-state button
- `JToggleButton.ToggleButtonModel`—It is a toggle button model
- `JToolBar`—It is a toolbar that is used for displaying commonly used controls
- `JToolBar.Separator`—It is a toolbar-specific separator
- `JToolTip`—It displays a tooltip for a component
- `JTree`—It displays a set of hierarchical data as an outline
- `JTree.DynamicUtilTreeNode`—It can wrap vectors/hashtables/arrays/strings and create appropriate children tree nodes
- `JTree.EmptySelectionModel`—It is a tree selection model that does not allow anything to be selected
- `JViewport`—It is the viewport through which you see the information
- `JWindow`—It is a window that can be displayed anywhere on the desktop

It can be seen in the preceding list that there is a Swing replacement for every AWT control and container except `Canvas`, the reason being that the `JPanel` class already supports all that the `Canvas` component did, so Java didn't find it necessary to add a separate `JCanvas` component.

For creating distinct types of UI elements, different classes are provided by the Swing API. This chapter discusses five classes that are part of a larger collection of classes—`JRootPane`, `JLayeredPane`, `JPanel`, `JFrame`, and `JMenuBar`. The Swing family tree is split in such a way that the `Container` class is divided into two—the group of classes derived from the `JComponent` class and another that is derived from the `Window` class, as shown in Figure 11.2:

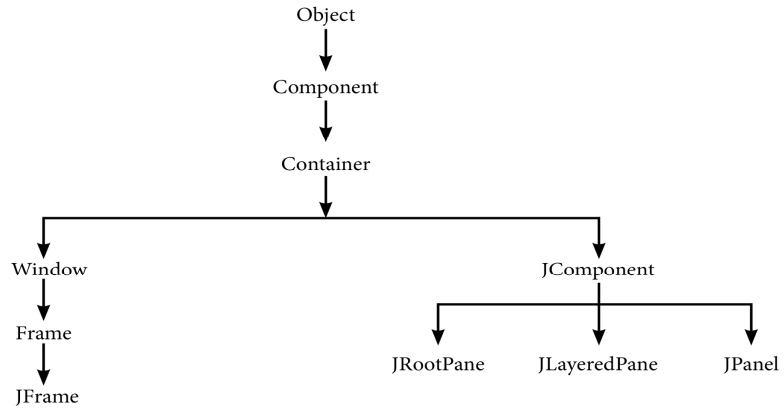


Figure 11.2: The Swing Family Tree

Heavyweight versus Lightweight Components

As there is a fundamental philosophical difference between the AWT and Swing UI components, it is not possible to add all the enhancements and new components to the AWT.

The AWT components are considered *heavyweight*. The reason being that every component in AWT takes up its window from the operating system environment in which it is running, thereby slowing down the performance of the program and consuming a lot of memory and system resources. On the other hand, a Swing component is

drawn simply as an image inside the container and does not take up window from the operating system environment. In addition, it consumes much less system resources. Therefore, the Swing components are considered *lightweight*. A few more significant points are as follows:

- All Swing components are derived from the `JComponent` class, and this class is, in turn, derived from the AWT `Container` class, which has no heavyweight window (called a *peer*) itself. Therefore, the `JComponent` is a lightweight class. You'll see in this chapter that the `JComponent class` adds a tremendous amount of programming support to the AWT Component class. Note that as some top level containers are not derived from the `JComponent` class, and the `JComponent` class is derived from the AWT `Container` class, all Swing components are also AWT components. You can also mix the AWT controls with the Swing controls in your programs. However, because Swing controls are merely drawn in their container, you can end up with strange results as AWT controls will appear on top of them.
- Not all Swing components are lightweight because lightweight components delete need the help of operating system to draw lightweight controls over the window. For that reason, Swing supports these heavyweight classes: `JFrame`, `JDialog`, `JWindow`, and `JApplet`.
- Just as you build AWT applets by using the `Applet` class and AWT applications by using the `Frame` class, you build Swing applets on the `JApplet` class and Swing applications by using the `JFrame` class. The fact that Swing is built on top of AWT is by no means seamless to the programmer. For example, to paint components, you don't override the `paint()` method anymore because Swing needs to do that to draw component borders and you have direct access to the parts of the program in which menus and dialog boxes are drawn. All in all, it takes some additional programming effort to move from AWT to Swing.

Swing Features

Apart from the large array of lightweight components in Swing, many other innovations have been introduced in it. Some of the major innovations are discussed as follows:

- **Borders**—You can draw borders in many different styles around components by using the `setBorder()` method.
- **Graphics debugging**—You can use the `setDebuggingGraphicsOptions()` method for setting up graphics debugging, which means that you can watch each line while drawing and flash it.
- **Easy mouseless operation**—It allows to easily connect keystrokes to components.
- **Tooltips**—You can use the `setToolTipText()` method of the `JComponent` class for providing tooltip to the components. The tooltips are small windows that are visible whenever you move the mouse over a component. They provide explanatory text.
- **Easy scrolling**—You can connect scrolling feature to different components; this feature was not possible in AWT.
- **Pluggable look and feel**—With the help of this feature, the appearance of applets and applications can be set to any one of the three standard appearances, such as Windows, Motif (Unix), or Metal (the standard Swing look).
- **New layout managers**—The `BoxLayout` and `OverlayLayout` layout managers are introduced in Swing.

Of these features, the pluggable look and feel is probably the most important because it allows you to select the appearance style of your program. In AWT, controls were built on windows from the operating platform and therefore, looked like other controls in that platform—something that gave AWT programs a very platform-specific appearance, which was not a good idea for a language that prides itself on being cross-platform. For example, as different fonts and controls are used in different platforms, the layout of programs could appear very different in different platforms. Swing introduces the Metal look, which is the new Java look, and it will appear the same across all platforms. In addition, you can give your programs a Windows or Motif look if you so choose. We'll take a look at this process in this chapter.

NOTE

In fact, Sun has created several other look and feel types that do not ship with the Software Development Kit (SDK), including one for the Apple and another platform-independent look and feel called Organic. In addition, several third parties have also created their own Java look and feels, such as JGoodies, Napkin, and Substance.

We have two essential programming differences between AWT and Swing programming to take a look at before entering into the "Immediate Solutions" section. The first difference is that you work with panes when creating Swing applets and applications, and the other is the Model View Controller programming architecture. We'll take a look at working with Swing panes first because you have to understand how panes work before setting them anywhere at all in Swing.

TIP

Here's another difference between AWT and Swing—when you redraw an item on any AWT screen, the `update()` *method is called to redraw the item's background. To avoid flickering, the programmers often override the* `update()` *method by directly calling the* `paint()` *method. On the other hand, case of the background of thean item is not redrawn by the* `update()` *method in case of Swing. This is because the Swing components can be transparent, instead the* `update()method` *just calls the* `paint()` method *directly.*

Graphics Programming Using Panes

The classes, which are the basis of applets and applications—`JApplet` and `JFrame`, have a child object that does all the graphics work, and this object is of the `JRootPane` class. In AWT programming, you can just add components directly to applets or frame windows; however, in Swing, it's a little more complex.

Here's the structure of the `JRootPane class`: This pane itself has two other panes—a layered pane and a glass pane. The glass pane is actually an AWT Component object that "rides" preceding everything else and intercepts mouse events when you make this pane visible (although it's usually transparent) and adds a mouse listener to it. The layered pane of the class `JLayeredPane` is where most of the action takes place. There are specific layers to which you have access that display menus, when they are opened, dialog boxes, and so on. From a programmer's point of view, the most interesting elements in the layered pane are probably the menu bar and content pane. The *content pane* is actually an object of the AWT `Container` class (there is no `JContentPane` class), and it's where the applets and applications display their components. In practice, this means that when you add components to your program, you add them to the content pane, which means you must first get a reference to that pane. If you have a menu bar in your program, it is supported with a `JMenuBar` object, which is displayed right above the content pane.

NOTE

Swing content panes use a border layout manager by default, whereas AWT applets use a flow layout by default, although AWT frame windows do use border layouts by default.

Model View Controller Architecture

The advantage of working with Swing is the pluggable look and feel feature that helps you in changing the appearance of a component. You can not only select from the look and feel options of a component but also create your own look and feel of the component. Such a pluggable look and feel of the Swing component has been assisted by designing the classes in a specific manner—the Model View Controller (MVC) architecture.

MVC architecture is the heart of Swing UI component programming, and an understanding of its terms is essential. Here's what these terms mean: the *model* of a component is where the component's data is stored, such as the state of a button or the items in a list. The *view* is the screen representation of the component, such as the way a button or list appears. Finally, the *controller* is that part of the component that handles input, such as mouse clicks. Figure 11.3 shows the MVC architecture:

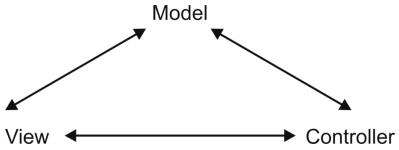


Figure 11.3: The MVC Architecture

It has become standard to divide larger programs into views and documents. For example, in Microsoft Visual C++, a view provides a window into a document, which is where the data in the program is stored, and multiple views can provide multiple windows into the document. Swing, inspired by the Smalltalk language, goes one step further and bases its UI components on the MVC architecture. As you can imagine, separating the view from the model is very useful in Swing, where the look and feel of components can change with a few lines of code. You'll learn more about MVC architecture soon. Now that you have an overview of Swing, let's move to the "Immediate Solutions" section.

Immediate Solutions

Working with Swing

The NP appears and says, "OK, I'm ready to start working with Swing. Where do I start?" "Well," you say, "probably with the `JComponent` class."

All Swing components are derived from the `JComponent` class. It is a lightweight class that is derived from the AWT `Container` class. The `JComponent` class is basically the `javax.swing.JComponent` class because the `javax` package includes Swing. The inheritance diagram for the `JComponent` class is as follows:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
```

The `JComponent` class adds a lot to the AWT `Container` class, such as the ability to draw pre-defined borders around UI components; add `PropertyChangeListener` objects, which are notified when a property's value changes; and much more. For understanding the fields, constructor, and methods of this large class, you can refer Tables 11.1, 11.2, and 11.3, respectively:

Table 11.1: Fields of the JComponent class

Field	Does this
`protected AccessibleContext accessibleContext`	AccessibleContext represents the minimum information that all accessible objects return. This information includes the accessible name, description, role, and state of the object, as well as information about its parent and children. This is associated with the `JComponent` class.
`protected EventListenerList listenerList`	It is a list of the current event listeners.
`static String TOOL_TIP_TEXT_KEY`	It is a comment to display when the cursor is over the component (also called value tip, flyover tip, or flyover label).
`protected ComponentUI ui`	It is the look and feel delegate for this component.
`static int UNDEFINED_CONDITION`	It is a constant used by some of the APIs to mean that no condition is defined.
`static int WHEN_ANCESTOR_OF_FOCUSED_ COMPONENT`	It is used for `registerKeyboardAction`, indicating that the command should be invoked when the receiving component is an ancestor of the focused component or is itself the focused component.
`static int WHEN_FOCUSED`	It is a constant used for `registerKeyboardAction`, indicating that the command should be invoked when the component has the focus.
`static int WHEN_IN_FOCUSED_WINDOW`	It is a constant used for `registerKeyboardAction`, indicating that the command should be invoked when the receiving component gets the focus or is itself the focused component.

Table 11.2: Constructor of the JComponent class

Constructor	Does this
`JComponent()`	The default `JComponent` constructor

Table 11.3: Methods of the JComponent class

Method	Does this
`void addAncestorListener (AncestorListener listener)`	It registers the listener so that it will receive `AncestorEvents` when it or any of its ancestors move or are made visible or invisible.
`void addNotify()`	It notifies that this component now has a parent component.

Table 11.3: Methods of the JComponent class

Method	Does this
`void addVetoableChangeListener (VetoableChangeListener listener)`	It adds a `VetoableChangeListener` to the listener list.
`void computeVisibleRect (Rectangle visibleRect)`	It gets the component's visible rectangle (the intersection of the visible rectangles for this component and all its ancestors).
`boolean contains (int x, int y)`	It defines the precise shape of this component for mouse processing.
`JToolTip createToolTip()`	It gets the instance of `JToolTip` that should be used to display the tooltip.
`void disable()`	It is deprecated and replaced by `setEnabled(boolean)` where the boolean value is false.
`void enable()`	It is deprecated and replaced by `setEnabled(boolean)` where the boolean value is true.
`void firePropertyChange (String propertyName, boolean oldValue, boolean newValue)`	It reports a bound property change for boolean properties.
`void firePropertyChange (String propertyName, char oldValue, char newValue)`	This method can be called when a bound property has changed, and it will send the appropriate `PropertyChangeEvent` to any registered `PropertyChangeListeners`.
`void firePropertyChange (String propertyName, int oldValue, int newValue)`	It reports a bound property change for integer properties.
`protected void fireVetoableChange(String propertyName, Object oldValue, Object newValue)`	It provides support for reporting constrained property changes.
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this `JComponent`.
`ActionListener getActionForKeyStroke (KeyStroke aKeyStroke)`	It gets the object that will perform the action registered for a given keystroke.
`ActionMap getActionMap()`	It returns the action map used for determining what action to fire for a particular `KeyStroke` binding.
`float getAlignmentX()`	It gets the vertical alignment.
`float getAlignmentY()`	It gets the horizontal alignment.
`AncestorListener[] getAncestorListeners()`	It returns an array of all ancestor listeners registered on this component.
`boolean getAutoscrolls()`	It gets the `autoscrolls` property.
`int getBaseLine(int width, int height)`	It returns the base line.
`Component.BaseLineResizeBehavior getBaselineResizeBehavior()`	It returns an enum, indicating how the baseline of the component changes as the size changes.
`Border getBorder()`	It gets the border of this component or null if no border is set.
`Rectangle getBounds (Rectangle rv)`	It stores the bounds of this component into return value `rv` and returns `rv`.
`Object getClientProperty (Object key)`	It gets the value of the property with the specified key.

Table 11.3: Methods of the JComponent class

Method	Does this
`protected Graphics getComponentGraphics (Graphics g)`	It gets the graphics object used to paint this component.
`JPopupMenu getComponentPopupMenu()`	It returns the `JPopupMenu` assigned for this component.
`int getConditionForKeyStroke (KeyStroke aKeyStroke)`	It gets the condition that determines whether a registered action occurs for the specified keystroke.
`int getDebugGraphicsOptions()`	It gets the state of graphics debugging.
`static Locale getDefaultLocale()`	It returns the default locale used to initialize the locale property of each `JComponent`.
`FontMetrics getFontMetrics(Font font)`	It gets the `FontMetrics` for the given `Font`.
`Graphics getGraphics()`	It gets this component's graphics context.
`int getHeight()`	It gets the current height of this component.
`boolean getInheritsPopupMenu()`	It returns whether the `JPopupMenu` should be inherited from the parent.
`InputMap getInputMap()`	It returns the input map that is used when the receiver has focus.
`InputMap getInputMap (int condition)`	It returns the input map that is used during the given condition.
`InputVerifier getInputVerifier()`	It returns the input verifier for this component.
`Insets getInsets()`	It gets the border's insets if a border has been set on this component. Otherwise, it calls `super.getInsets()`.
`Insets getInsets (Insets insets)`	It gets an `Insets` object containing this component's inset values.
`<T extends EventListener>T[] getListeners(Class<T> listenerType)`	It returns an array of all the listeners currently registered as *Foo*`Listeners` to the `JComponent`.
`Point getLocation(Point rv)`	It stores the `x,y` origin of this component into return value `rv` and returns `rv`.
`Dimension getMaximumSize()`	It gets the component's maximum size.
`Dimension getMinimumSize()`	It gets the component's minimum size.
`Component getNextFocusableComponent()`	It is deprecated and replaced by `FocusTraversalPolicy`.
`Point getPopupLocation (MouseEvent event)`	It returns the preferred location to display the pop-up menu in this component's coordinate system.
`Dimension getPreferredSize()`	If the preferred size has been set to a non-null value, this method returns it.
`KeyStroke[] getRegisteredKeyStrokes()`	It gets the keystrokes that will initiate registered actions.
`JRootPane getRootPane()`	It gets the `JRootPane` ancestor for a component.
`Dimension getSize (Dimension rv)`	It stores the width/height of this component into return value `rv` and returns `rv`.
`Point getToolTipLocation (MouseEvent event)`	It gets the tooltip location in the component coordinate system.
`String getToolTipText()`	It gets the tooltip string that has been set with `setToolTipText()`.
`String getToolTipText (MouseEvent event)`	It gets the string to be used as the tooltip for the event.

Table 11.3: Methods of the JComponent class

Method	Does this
`Container getTopLevelAncestor()`	It gets the top-level ancestor of this component or null if this component has not been added to any container.
`TransferHandler getTransferHandler()`	It gets the `transferHandler` property.
`String getUIClassID()`	It gets the `UIDefaults` key used to find the name of the `swing.plaf.ComponentUI` class.
`boolean getVerifyInputWhenFocusTarget()`	It indicates whether the input verifier for the current focus owner will be called before this component requests focus.
`VetoableChangeListener[] getVetoableChangeListeners ()`	It returns an array of all vetoable change listeners registered on this component.
`Rectangle getVisibleRect()`	It gets the component's visible rectangle.
`int getWidth()`	It gets the current width of this component.
`int getX()`	It gets the current x coordinate of this component's origin.
`int getY()`	It gets the current y coordinate of this component's origin.
`void grabFocus()`	It requests that this component gets the input focus and its top level ancestor becomes the focused window.
`boolean isDoubleBuffered()`	It indicates whether this component should use a buffer to paint.
`static boolean isLightweightComponent (Component c)`	It returns True when this component is lightweight (means it does not have a native window system peer).
`boolean isManagingFocus()`	It is deprecated and replaced by `Component.setFocusTraversalKeys(int, Set)` and `Component.setFocusCycleRoot(boolean)`.
`boolean isOpaque()`	It returns True if this component is completely opaque.
`boolean isOptimizedDrawingEnabled()`	It returns True if this component tiles its children, means it guarantees non-overlapping of its components.
`boolean isPaintingForPrint()`	It returns True if the current painting operation on this component is a part of the `print` operation.
`protected boolean isPaintingOrigin()`	It returns True if a paint operation triggered on a child component should cause painting to originate from this component or one of its ancestors.
`boolean isPaintingTile()`	It returns True if the component is currently painting a tile.
`boolean isRequestFocusEnabled()`	It returns True if the receiving component should get the focus.
`boolean isValidateRoot()`	If this method returns True, `revalidate` calls made by descendants of this component will cause the entire tree beginning with this root to be validated.
`void paint(Graphics g)`	It is invoked by Swing to draw components to the specified graphics.
`protected void paintBorder(Graphics g)`	It paints the component's border.
`protected void paintChildren(Graphics g)`	It paints this component's children.
`protected void paintComponent(Graphics g)`	It calls the UI delegate's `paint` method if the UI delegate is non-null.
`void paintImmediately (int x, int y, int w, int h)`	It paints immediately the specified region of this component and all of its descendants that overlap the region.
`void paintImmediately (Rectangle r)`	It paints the specified region immediately.
`protected String paramString()`	It gets a string representation of this component.

Table 11.3: Methods of the JComponent class

Method	Does this
`void print(Graphics g)`	It invokes this method to print the component to the graphics specified.
`void printAll (Graphics g)`	It prints the component.
`protected void printBorder (Graphics g)`	It prints the component's border.
`protected void printChildren(Graphics g)`	It prints this component's children.
`protected void printComponent(Graphics g)`	It is invoked during a printing operation.
`protected void processComponentKeyEvent (KeyEvent e)`	It processes any key events recognized by the component itself.
`protected void processKeyBinding(KeyStroke ks, KeyEvent e, int condition, boolean pressed)`	It processes the key bindings for `KeyStroke ks` as a consequence of the `KeyEvent e`.
`protected void processKeyEvent(KeyEvent e)`	It overrides `processKeyEvent` to process events.
`protected void processMouseEvent (MouseEvent e)`	It processes mouse events occurring on this component by sending them to any registered `MouseListener` objects.
`protected void processMouseMotionEvent (MouseEvent e)`	It processes mouse motion events.
`void putClientProperty (Object key, Object value)`	It adds an arbitrary key/value client property to this component.
`void registerKeyboardAction (ActionListener anAction, KeyStroke aKeyStroke, int aCondition)`	It registers a new keyboard action. `anAction` will be invoked if a key event matching `aKeyStroke` occurs and `aCondition` is verified. The `KeyStroke` object defines a particular combination of a keyboard key and one or more modifiers (alt, shift, ctrl, or meta).
`void registerKeyboardAction(ActionListener anAction, String aCommand, KeyStroke aKeyStroke, int aCondition)`	It registers a new keyboard action. `anAction` will be invoked if a key event matching `aKeyStroke` occurs and `aCondition` is verified. The `KeyStroke` object defines a particular combination of a keyboard key and one or more modifiers (alt, shift, ctrl, meta). The `aCommand` will be set in the delivered event if specified.
`void removeAncestorListener (AncestorListener listener)`	It unregisters the given listener so that it will no longer get `AncestorEvents`.
`void removeNotify()`	It indicates a notification that this component no longer has a parent component.
`void removeVetoableChangeListener (VetoableChangeListener listener)`	It removes a `VetoableChangeListener` object from the listener list.
`void repaint(long tm, int x, int y, int width, int height)`	It adds the specified region to the dirty region list if the component is showing. It takes following parameters: 1. tm - maximum time in milliseconds before update 2. x - the x coordinate 3. y - the y coordinate 4. width - the width 5. height - the height

Table 11.3: Methods of the JComponent class

Method	Does this
`void repaint(long tm)`	It repaints the component. If this component is a lightweight component, it would result in a call to paint within tm milliseconds.
`boolean requestDefaultFocus()`	It is deprecated and replaced by the `FocusTraversalPolicy.getDefaultComponent (Container).requestFocus()` method.
`void requestFocus()`	It requests that this component gets the input focus.
`void requestFocus(boolean temporary)`	It requests that this component gets the input focus.
`void requestFocusInWindow()`	It requests that this component gets the input focus.
`void requestFocusInWindow (boolean temporary)`	It is a parameterized form of the `requestFocusInWindow()` method that requests that this component gets the input focus. Its parameter is a boolean temporary; temporary will be true if the focus change is temporary, such as when the window loses the focus.
`void resetKeyboardActions()`	It unregisters all keyboard actions.
`void reshape(int x, int y, int w, int h)`	It is deprecated and replaced by the `Component.setBounds(int, int, int, int)` method.
`void revalidate()`	It provides support for deferred automatic layout.
`void scrollRectToVisible (Rectangle aRect)`	It forwards the `scrollRectToVisible` message to the `JComponent's` parent.
`void setActionMap (ActionMap am)`	It sets the action map to `am`.
`void setAlignmentX (float alignmentX)`	It sets the vertical alignment.
`void setAlignmentY (float alignmentY)`	It sets the horizontal alignment.
`void setAutoscrolls(boolean autoscrolls)`	It sets the `autoscrolls` property.
`void setBackground (Color bg)`	It sets the background color of this component.
`void setBorder (Border border)`	It sets the border of this component.
`void setDebugGraphicsOptions (int debugOptions)`	It enables or disables diagnostic information about every graphics operation performed within the component or one of its children.
`void setDefaultLocale(Locale l)`	It sets the default locale used to initialize the locale property of each `JComponent class`.
`void setDoubleBuffered (boolean aFlag)`	It sets whether this receiving component should use a buffer to paint.
`void setEnabled (boolean enabled)`	It sets whether this component is enabled.
`void setFocusTraversalKeys (int id, Set<? extends AWTKeyStroke> keystrokes)`	It sets the focus traversal keys for a given traversal operation for this component.
`void setFont(Font font)`	It sets the font for this component.
`void setForeground (Color fg)`	It sets the foreground color of this component.
`void setInheritsPopupMenu (boolean value)`	It sets whether or not `getComponentPopupMenu` should delegate to the parent if this component has no `JPopupMenu` assigned.
`void setInputMap(int condition, InputMap map)`	It sets the `InputMap` to use under the condition `condition`.

Table 11.3: Methods of the JComponent class	
Method	**Does this**
`void setInputVerifier (InputVerifier inputVerifier)`	It sets the input verifier for this component.
`void setMaximumSize (Dimension maximumSize)`	It sets the maximum size of this component to a constant value.
`void setMinimumSize (Dimension minimumSize)`	It sets the minimum size of this component to a constant value.
`void setNextFocusableComponent (Component aComponent)`	It is deprecated and replaced by `FocusTraversalPolicy`.
`void setOpaque (boolean isOpaque)`	It enables or disables the component to paint every pixel within its bounds.
`void setPreferredSize (Dimension preferredSize)`	It sets the preferred size of this component.
`void setRequestFocusEnabled (boolean aFlag)`	It sets whether the component can get the focus.
`void setToolTipText (String text)`	It registers the text to be displayed in a tooltip.
`void setTransferHandler (TransferHandler newHandler)`	It sets the `transferHandler` to transfer data in and out of this component.
`protected void setUI (ComponentUI newUI)`	It sets the look and feel delegated for this component.
`void setVerifyInputWhenFocusTarget(boolean verifyInputWhenFocusTarget)`	It sets the value to indicate whether input verifier for the current focus owner will be called before this component requests focus.
`void setVisible (boolean aFlag)`	It makes the component visible or invisible.
`void unregisterKeyboardAction (KeyStroke aKeyStroke)`	It is an obsolete method.
`void update(Graphics g)`	It calls the `paint()` method.
`void updateUI()`	It resets the UI property to a value from the current look and feel.

Preparing to Create a Swing Applet

"OK," the NP "I'm ready to start creating applets by using Swing. How do I proceed?" "With the `JApplet` class," you say. "Pull up a chair and we'll go through it."

The `JApplet` class is a heavyweight container, which is the foundation of Swing applets and is an extension of the AWT `Applet` class. The `JApplet` class's richness in functionality is not found in `Applet`; for example, `JApplet` supports various panes, such as content pane, glass pane, and layered pane.

Before Java SE 5, you could not easily invoke the `add()` method to an instance of the `JApplet` class in order to add a component. To do so, you had to call the `add()` method to the content pane of the `JApplet` object. The content pane was obtained by the `Container`'s `getContentPane`() method, and the `add()` method of `Container` was then used to add a component to a content pane, as shown in the following code:

```
Container c = getContentPane();
c.add(someComponent);
```

You could invoke the `remove()` and `setLayout()` methods in the same way. But now, you can directly call the `add()`, `remove()`, and `setLayout()` methods on the `JApplet` class. The `JApplet` class has one child—an object of the `JRootPane` class—and that's where the drawing goes on as well as where you add components. The inheritance diagram for the `JApplet` class is as follows:

```
java.lang.Object
  |____java.awt.Component
```

```
            |____java.awt.Container
                  |____java.awt.Panel
                        |____java.applet.Applet
                              |____javax.swing.JApplet
```

You can find the fields of the `JApplet` class in Table 11.4, its constructor in Table 11.5, and the methods in Table 11.6:

Table 11.4: Fields of the JApplet class

Field	Does this
`protected AccessibleContext accessibleContext`	It is an accessibility context
`protected JRootPane rootPane`	It is the root pane
`protected boolean rootPaneCheckingEnabled`	It indicates whether calls to the `add` and `setLayout` methods are forwarded to the `contentPane`

Table 11.5: Constructor of the JApplet class

Constructor	Does this
`JApplet()`	It creates a Swing applet instance

Table 11.6: Methods of the JApplet class

Method	Does this
`protected void addImpl (Component comp, Object constraints, int index)`	It adds the given child component to the content pane
`protected JRootPane createRootPane()`	It is called by the constructor methods to create the default root pane
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this `JApplet`
`Container getContentPane()`	It gets the `contentPane` object for this applet
`Component getGlassPane()`	It gets the `glassPane` object for this applet
`Graphics getGraphics()`	It gets a graphics context for this applet
`JMenuBar getJMenuBar()`	It gets the menu bar set on this applet
`JLayeredPane getLayeredPane()`	It gets the `layeredPane` object for this applet
`JRootPane getRootPane()`	It gets the `rootPane` object for this applet
`TransferHandler getTransferHandler()`	It gets the `transferHandler` property
`protected boolean isRootPaneCheckingEnabled ()`	It It checks whether calls to the `add` and `setLayout methods` are forwarded to the `contentPane`
`protected String paramString()`	It gets a string representation of this `JApplet` class
`void remove (Component comp)`	It removes the specified component from this container
`void repaint(long time,int x,int y, int width,int height)`	It repaints the specified rectangle of this component within `time` milliseconds
`void setContentPane (Container contentPane)`	It sets the `contentPane` property
`void setGlassPane (Component glassPane)`	It sets the `glassPane` property
`void setJMenuBar (JMenuBar menuBar)`	It sets the menu bar for this applet

Table 11.6: Methods of the JApplet class	
Method	**Does this**
`void setLayeredPane (JLayeredPane layeredPane)`	It sets the `layeredPane` property
`void setLayout (LayoutManager manager)`	It sets the layout of its content pane instead
`protected void setRootPane (JRootPane root)`	It sets the `rootPane` property
`protected void setRootPaneCheckingEnabled (boolean enabled)`	It sets whether calls to the `add` and `setLayout` method are forwarded to the `contentPane`
`void setTransferHandler (TransferHandler newHandler)`	It sets the `transferHandler` property
`void update(Graphics g)`	It calls the `paint(g)` method and does not redraw the background

To start a Swing applet, you just derive a class from the `JApplet` class as follows (note that in this example, we import `javax.swing.*` in order to use `JApplet`):

```
import java.awt.*;
import javax.swing.*;
/* <APPLET
   CODE=SwingApplet.class
   WIDTH=300
   HEIGHT=200 >
</APPLET>
*/
public class SwingApplet extends JApplet { ... }
```

Unfortunately, things get more complex at this point (when you want to draw in an applet or add controls to it) because you normally work with the content pane in the applet rather than directly with the applet itself. Both the `JApplet` and `JFrame` objects have one child object—an object of the class `JRootPane`—and the content pane is a part of the root pane. We'll take a look at root panes in the next solution.

Understanding Root Panes

The `JRootPane` class is the class that manages the appearance of the `JApplet` and `JFrame` objects, and it is worth taking a look at what's in the `JRootPane` class because all its functionality is available to you in applets and applications. In particular, the root pane contains the glass pane, content pane, and layered pane, which you'll see a lot in Swing programming. The inheritance diagram for the JRootPane class is as follows:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
                |____javax.swing.JComponent
                        |____javax.swing.JRootPane
```

You'll find the fields of the `JRootPane` class in Table 11.7, its constructor in Table 11.8, and the methods in Table 11.9:

Table 11.7: Fields of the JRootPane class	
Field	**Does this**
`static int COLOR_CHOOSER_DIALOG`	It is the constant used for the `windowDecorationStyle` property
`protected Container contentPane`	It is the content pane used to hold the elements
`protected JButton defaultButton`	It is the button that gets activated when the pane has the focus and a UI-specific action, such as pressing the Enter key, occurs

Table 11.7: Fields of the JRootPane class	
Field	**Does this**
`protected javax.swing.JRootPane.DefaultAction defaultPressAction`	It is deprecated and represented the action to be taken when the default button is pressed
`protected javax.swing.JRootPane.DefaultAction defaultReleaseAction`	It is deprecated and represented the action to be taken when the default button is released
`static int ERROR_DIALOG`	It is the constant used for the `windowDecorationStyle` property to display the error message
`static int FILE_CHOOSER_DIALOG`	It is the constant used for the `windowDecorationStyle` property to display the dialog box displaying the file to be chosen
`static int FRAME`	It is the constant used for the `windowDecorationStyle` property to display the frame
`protected Component glassPane`	It is the glass pane that overlays the menu bar and content pane so that it can intercept the mouse movements
`static int INFORMATION_DIALOG`	It is the constant used for the `windowDecorationStyle` property
`protected JLayeredPane layeredPane`	It is the layered pane that manages the menu bar and content pane
`protected JMenuBar menuBar`	It is the menu bar that contains different menus and submenus
`static int NONE`	It is the constant used for the `windowDecorationStyle` property
`static int PLAIN_DIALOG`	It is the constant used for the `windowDecorationStyle` property to display the plain dialog box
`static int QUESTION_DIALOG`	It is the constant used for the `windowDecorationStyle` property to display the question dialog box
`static int WARNING_DIALOG`	It is the constant used for the `windowDecorationStyle` property to display the warning message

Table 11.8: The Constructor of the JRootPane class	
Constructor	**Does this**
`JRootPane()`	It creates a `JRootPane` component, setting up its `glassPane`, `LayeredPane`, and `contentPane` components

Table 11.9: Methods of the JRootPane class	
Method	**Does this**
`protected void addImpl (Component comp, Object constraints, int index)`	It is overridden to ensure the position of the glass component as the zero child
`void addNotify()`	It notifies this component that it now has a parent component
`protected Container createContentPane()`	It is called by the constructor methods to create the default `contentPane`
`protected Component createGlassPane()`	It is called by the constructor methods to create the default `glassPane`
`protected JlayeredPane createLayeredPane()`	It is called by the constructor methods to create the default `layeredPane`
`protected LayoutManager createRootLayout()`	It is called by the constructor methods to create the default `layoutManager`
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this `JRootPane`
`Container getContentPane()`	It gets the content pane

Table 11.9: Methods of the JRootPane class	
Method	**Does this**
`JButton getDefaultButton()`	It gets the value of the `defaultButton` property
`Component getGlassPane()`	It gets the current glass pane for this `JRootPane`
`JMenuBar getJMenuBar()`	It gets the menu bar from the layered pane
`JLayeredPane getLayeredPane()`	It gets the layered pane used by the root pane
`JMenuBar getMenuBar()`	It is deprecated and replaced by `getJMenuBar()`
`RootPaneUI getUI()`	It returns the `LookAndFeel` object that renders this component
`String getUIClassID()`	It returns a string that specifies the name of the look and feel class that renders this component
`int getWindowDecorationStyle()`	It returns a constant that specifies the Window decorations provided by the `JRootPane`
`boolean isOptimizedDrawingEnabled ()`	The `glassPane` and `contentPane` have the same bounds, which mean `JRootPane` does not tile its children, and this should return False
`boolean isValidateRoot()`	If a descendant of this `JRootPane` calls `revalidate`, validation will occur from here on down
`protected String paramString()`	It gets a string representation of the root pane
`void removeNotify()`	It notifies the active component that it longer has a parent component
`void setContentPane (Container content)`	It sets the content pane
`void setDefaultButton (JButton defaultButton)`	It sets the `defaultButton` property
`void setDoubleBuffered (boolean aFlag)`	It sets whether this component should use a buffer to paint
`void setGlassPane (Component glass)`	It sets a specified component to be the glass pane for this root pane
`void setJMenuBar (JMenuBar menu)`	It adds or changes the menu bar used in the layered pane
`void setLayeredPane (JLayeredPane layered)`	It sets the layered pane for the root pane
`void setMenuBar (JMenuBar menu)`	It is deprecated and replaced by `setJMenuBar(JMenuBar menu)`
`void setUI(RootPaneUI ui)`	It sets the `LookAndFeel` object that renders this component
`void setWindowDecorationStyle (int windowDecorationStyle)`	It sets the type of window decorations the `JRootPane` should provide
`void updateUI()`	It resets the UI property to a value from the current look and feel

As far as construction of applets and applications is concerned, you usually work with the content pane of the root pane. The content pane itself, however, is a part of another pane—the layered pane. We'll take a look at layered panes in the next solution.

Understanding Layered Panes

The layered pane inside the root pane holds the actual components that appear in applets and applications, including the menu bars and the content pane. The inheritance diagram for the layered pane class, that is, JLayeredPane is as follows:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.JLayeredPane
```

For convenience, JLayeredPane divides the depth range into the following different layers. Programmer can put the component on any of the following layers, making it easy to ensure that components overlap properly, without having to worry about the specific depths:

- DEFAULT_LAYER—The standard, bottommost layer where most components go
- PALETTE_LAYER—The palette layer resides above the default layer and is beneficial for floating toolbars and palettes
- MODAL_LAYER—The layer used for modal dialog boxes
- POPUP_LAYER—The pop-up layer is displayed above the dialogs
- DRAG_LAYER—When you drag a component, assigning it to the drag layer makes sure that it is positioned over all other components in the container

You can use the JLayeredPane methods—moveToFront(), moveToBack(), and setPosition()—to reposition a component within its layer. You can also use the setLayer() method for changing the current layer of the component. You'll find the fields of the JLayeredPane class in Table 11.10, its constructor in Table 11.11, and its methods in Table 11.12:

Table 11.10: Fields of the JLayeredPane class

Field	Does this
static Integer DEFAULT_LAYER	It is a convenience object defining the Default layer
static Integer DRAG_LAYER	It is a convenience object defining the Drag layer
static Integer FRAME_CONTENT_LAYER	It is a convenience object defining the Frame Content layer
static String LAYER_PROPERTY	It is a bound property
static Integer MODAL_LAYER	It is a convenience object defining the Modal layer
static Integer PALETTE_LAYER	It is a convenience object defining the Palette layer
static Integer POPUP_LAYER	It is a convenience object defining the Popup layer

Table 11.11: Constructor of the JLayeredPane class

Constructor	Does this
JLayeredPane()	It creates a new JLayeredPane object

Table 11.12: Methods of the JLayeredPane class

Method	Does this
protected void addImpl (Component comp, Object constraints, int index)	It adds the indicated component to this container at the indicated index
AccessibleContext getAccessibleContext()	It gets the AccessibleContext associated with this JLayeredPane
int getComponentCountInLayer (int layer)	It gets the number of children in the indicated layer
Component[] getComponentsInLayer (int layer)	It gets an array of the components in the indicated layer
protected Hashtable<Component, Integer> getComponentToLayer()	It gets the hashtable that maps components to layers
int getIndexOf(Component c)	It gets the index of the indicated component
int getLayer(Component c)	It gets the layer attribute for the indicated component
static int getLayer (JComponent c)	It gets the layer property for a JComponent and does not cause any side effects like setLayer()

Table 11.12: Methods of the JLayeredPane class

Method	Does this
`static JLayeredPane getLayeredPaneAbove (Component c)`	It gets the first `JLayeredPane` that contains the indicated component
`protected Integer getObjectForLayer (int layer)`	It gets the `Integer` object associated with an indicated layer
`int getPosition (Component c)`	It gets the relative position of the component within its layer
`int highestLayer()`	It gets the highest layer value from all current children
`protected int insertIndexForLayer (int layer, int position)`	It determines the proper location to insert a new child based on the layer and position requests
`boolean isOptimizedDrawingEnabled()`	It returns False if components in the pane can overlap, which makes optimized drawing possible
`int lowestLayer()`	It gets the lowest layer value from all current children
`void moveToBack (Component c)`	It moves the component to the bottom of the components in its current layer
`void moveToFront (Component c)`	It moves the component to the top of the components in its current layer
`void paint(Graphics g)`	It paints the layered pane within the indicated graphics context
`protected String paramString()`	It gets a string representation of this `JLayeredPane`
`static void putLayer (JComponent c, int layer)`	It sets the layer property on a `JComponent`
`void remove(int index)`	It removes the indexed component from this pane
`void removeAll()`	It removes all components from this pane
`void setLayer (Component c, int layer)`	It sets the layer attribute on the indicated component, making it the bottommost component in that layer
`void setLayer(Component c, int layer, int position)`	It sets the layer attribute for the indicated component and also sets its position within that layer
`void setPosition(Component c, int position)`	It moves the component to `position` within its current layer

As far as typical drawing operations go in applets and applications, the most important part of the layered pane is the content pane, which we'll discuss in the following section.

Understanding Content Panes

You usually place controls and graphics in the content pane of an applet or application. You may be expecting the content pane to be supported by its own class, but in fact, a content pane is just an AWT Container object (which is itself a lightweight class). You can get access to the content pane in the layered pane of an applet or application by using the `getContentPane()` method of the `JApplet` and `JFrame` classes.

The importance of the content pane is that it is where you usually add components to applets and applications as well as where you draw graphics; this is a major difference from AWT programming. Another important point to know is that content panes use border layouts by default. To see how this actually works in a code, take a look at the next solution.

Working with Content Panes

The earlier versions of Java SE had some limitations with the mode of addition of components, like `JFrame`, `JDialog`, `JWindow`, `JApplet`, and `JInternalFrame`, to the top-level Swing containers; however, after Java SE 5, it has become easier. For example, initially, if any component, `someComponent`, had to be added to

`JFrame` named `myFrame`, there was a need to call the `getContentPane()` method of `JFrame` foradding the component as follows:

```
myFrame.getContentPane().add(someComponent)
```

But, after Java SE 5, the addition of a new component has become feasible without any such hindrances. Now, it is also possible to directly set the layout manager and remove components from the container. This section would enlighten upon how this is achieved, i.e., how can `add()`, `setLayout()`, and `remove()` be called directly without calling the `getContentPane()method` and why can't `ContentPane` be ignored even though you no longer have to call it to add components.

Suppose you have a `JFrame` for which you would like to set the layout manager and add various components. If you refer the `JFrame` documentation, it can be seen that `JFrame` is inheriting the `add()` method from `java.awt.Container`. The `add()` method is used to add a component to the `JFrame` container. The `Container` class also includes the `setLayout()` method, which is used to specify the layout manager for the given container, as shown in the following code:

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import java.awt.GridLayout;
import java.awt.Color;
public class JFrameDemo extends JFrame {
JFrameDemo() {
super("JFrameDemo");
setBackground(Color.BLACK);
setLayout(new GridLayout(7,7,2,2));
for(int i=0;i<49; i++) {
   JPanel jpanel1 = new JPanel();
   jpanel1.setBackground(new Color (
   (int)(Math.random()*16777215)));
   add(jpanel1); }
   setSize(400,400);
   setVisible(true); }
         public static void main(String[] args){ new JFrameDemo();}
   }
```

Compile and run the code of `JFrameDemo.java`. With this, a grid of forty-nine `JPanel` is found whose background colors are assigned randomly to the `JPanel`, as shown in Figure 11.4:

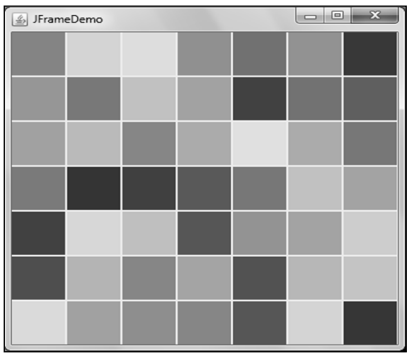


Figure 11.4: JFrame Showing the Grid

Earlier to Java SE 5, you had to explicitly call the `getContentPane()` method and set the `add()` or `setLayout()` method on the `Container` object returned by the `getContentPane()method`, as shown in the following code:

```
getContentPane().setLayout(new GridLayout(7, 7, 2, 2));
getContentPane().add(jpanel1);
```

But now, you can call the `add()` or `setLayout()` method directly on the `Container` object. You might notice a subtle problem. The following line in the `JFrameDemo` program does not seem to have any effect:

```
setBackground(Color.BLACK);
```

Note that you can write the color name as in `Color.BLACK` in both the uppercase and lowercase. It might be expected at this stage to have the background of the `JFrame` to be colored black so that the grid lines in between the colored `JPanel` are black and not grey. The background of the `JFrame` is not ordinarily visible although it is colored black because the color of the content pane is also black and resides on top of the region. If resizing of the `JFrame` is done at this time, you might momentarily see the black background. Instead, change the `setBackground()` line in the `JFrameDemo` program to the following:

```
getContentPane().setBackground(Color.BLACK);
```

Now, the grid lines are colored black, as shown in Figure 11.5:

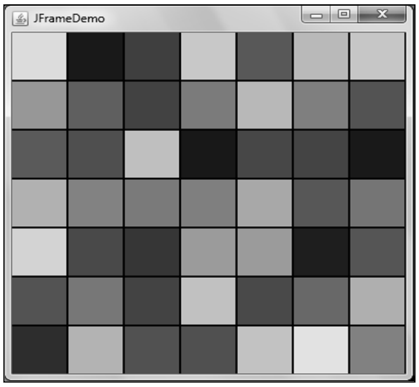


Figure 11.5: Showing the Grid

The colors of `JPanel` will differ in different runs as the color is being generated randomly.

The purpose of explaining this at this point of time is simply to understand that although you do not need to explicitly call the `getContentPane()` method when using the `add()`, `remove()`, and `setLayout()` methods in `JFrame`, you cannot still ignore the role of the content pane.

Creating a Swing Applet

"OK," says the NP, "I'm ready to create a Swing applet and add items to the content pane. How does it work?" "Sit down and let's discuss it," you reply.

Painting in Swing vs. AWT

If you have gone through the AWT concepts earlier in this book, you might be expecting a `paint()` method that you can use to handle painting operations and graphics in Swing applets and applications. In fact, there is a `paint()` method, but you should not override it unless you really know what you are doing because Swing itself uses that method to draw borders around the component as well as to draw any child components inside the component and other tasks.

Instead of using the `paint()` method for painting operations, you can now use the `paintComponent()` method. This has some ramifications for you when creating applets and applications compared to AWT programming. Instead of just overriding the applet or frame window class's `paint()` method, you now have to create a component to do your painting in because the content pane in a program is not a class that you can override methods in—it's an object (one alternative is to create a custom content pane class and use the `setContentPane()` method to install it in your program). In addition, note that when you work with the `paintComponent()` method, the first thing you should do is to call `super.paintComponent()`, the super class's `paintComponent()` method, to let the super class do additional painting.

Displaying Controls in Swing Vs AWT

To add controls to a content pane, you use the content pane's `add()` method. The content pane in both applets and applications use a border layout by default, unlike AWT programming where applets and frame windows use flow layouts and border layouts, respectively, by default.

To add controls to a content pane, you first set the layout manager to whatever you like or use the default border layout. To actually add the controls, you use the `add()` method, just as you do in AWT programming.

Let's take an example. In this case, we'll just display the text "Kogent Learning Solutions!" in an applet. How does that text get displayed? We can't just display it in the default content pane of an applet because that pane is an object, not a class. Therefore, we can't override its `paintComponent()` method. On the other hand, we can add a panel, as represented by the `JPanel` class, to the applet so that it covers the content pane.

We can also draw in that panel by overriding its `paintComponent()` method (note that we could create a new content pane class, override its `paintComponent()` method, and install it with the `setContentPane()` method, but `JPanel` is generally used to create content panes anyway).

TIP

If you ever want to find out exactly what values were used to construct a visual component, such as JApplet, give it the focus, press the Shift+Ctrl+F1 keys, and watch the console window for a complete overview.

Using the *JPanel* Class

The `JPanel` class is the all-purpose container class of Swing, and it's an important one to know about. The inheritance diagram for the `JPanel` class is as follows:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.JPanel
```

Unlike the heavyweight Swing containers, the `JPanel` class uses a flow layout by default. You'll find the constructors of the `JPanel` class in Table 11.13 and its methods in Table 11.14:

Table 11.13: Constructors of the JPanel class

Constructor	Does this
`JPanel()`	It constructs a new `JPanel` with a double buffer and a flow layout
`JPanel(boolean isDoubleBuffered)`	It constructs a new `JPanel` with a flow layout and the specified buffering strategy
`JPanel(LayoutManager layout)`	It constructs a new buffered `JPanel` with the specified layout manager
`JPanel(LayoutManager layout, boolean isDoubleBuffered)`	It constructs a new `JPanel` with the specified layout manager and buffering strategy

Table 11.14: Methods of the JPanel class

Method	Does this
`AccessibleContext getAccessibleContext()`	It gets the accessible context
`panelUI getUI()`	It returns the look and feel object that renders this component
`String getUIClassID()`	It gets a string that specifies the name of the "look and feel" class that renders this component
`protected String paramString()`	It gets a string representation of the `JPanel`
`void setUI(PanelUI ui)`	It sets the look and feel object that renders this component
`void updateUI()`	It resets the UI property with a value from the current look and feel

The following code shows how we create a `JPanel` class that displays the text "Kogent Learning Solutions!" (Note that we are giving this panel an orange background and calling the `paintComponent()` method; the `paintComponent()` method is passed an AWT Graphics object, which we can use to draw the text):

```
class TextPanel extends JPanel {
TextPanel() { setBackground(Color.orange); }
public void paintComponent (Graphics g) {
super.paintComponent(g);
```

```
    g.drawString("Kogent Learning Solutions!", 50, 80); }
    }
```

Now, we can add an object of this new class to the content pane of an applet by using the `JApplet` class's `getContentPane()` method and then use the content pane's `add()` method to actually add the panel, as shown in the following code:

```
import java.awt.*;
import javax.swing.*;
/*<APPLET
  CODE=JPanelDemo.class
  WIDTH=250
  HEIGHT=200>
</APPLET>*/
public class JPanelDemo extends JApplet {
  TextPanel panel;
  public void init() {
        Container c = getContentPane();
        panel = new TextPanel();
        c.add(panel); }
  }
```

That's all it takes—now, we can compile and run this applet, as shown in Figure 11.6:



Figure 11.6: Using a JPanel

Creating a Swing Application

"OK," says the NP, "I'm ready to create an application using Swing. Where do I start?" "By creating a frame window," you say. "Using the `Frame` class?" the NP asks. "Nope," you say, "the `JFrame` class."

In AWT applications, you usually use a frame window; similarly, while creating applications in Swing, you use a `JFrame` object, which has a root pane child. The inheritance diagram for the `JFrame` class is as follows (note that it's derived from the `Frame` class and is therefore a heavyweight container):

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____java.awt.Window
                    |____java.awt.Frame
                          |____javax.swing.JFrame
```

Note that the default layout in a `JFrame` component is a border layout. You'll find the fields of the `JFrame` class in Table 11.15, its constructors in Table 11.16, and its methods in Table 11.17:

Table 11.15: Fields of the JFrame class

Field	Does this
`protected AccessibleContext accessibleContext`	The accessible context property used to access the context of frame
`static int EXIT_ON_CLOSE`	Exit the application

Table 11.15: Fields of the JFrame class	
Field	**Does this**
`protected JRootPane rootPane`	The `JRootPane` that manages the `contentPane`, optional menu bar, and glass pane
`protected boolean rootPaneCheckingEnabled`	If True, then calls to `add` and `setLayout` are forwarded to the `contentPane`

Table 11.16: Constructors of the JFrame class	
Constructor	**Does this**
`JFrame()`	It constructs an initially invisible new frame
`JFrame (GraphicsConfiguration gc)`	It constructs a new frame with the specified `GraphicsConfiguration gc` of a screen device and blank title
`JFrame(String title)`	It constructs a new frame with the specified title
`JFrame(String title, GraphicsConfiguration gc)`	It creates a new frame with the specified title and the `GraphicsConfiguration` of a screen device

Table 11.17: Methods of the JFrame class	
Method	**Does this**
`protected void addImpl (Component comp, Object constraints, int index)`	It adds the indicated child component to the content pane
`protected JRootPane createRootPane()`	It is called by the constructor methods to create the default root pane
`protected void frameInit()`	It is called by the constructors to initialize the `JFrame` properly
`AccessibleContext getAccessibleContext()`	It gets the accessible context associated with this `JFrame`
`Container getContentPane()`	It gets the `contentPane` object for this frame
`int getDefaultCloseOperation()`	It gets the operation that occurs when the user closes the frame
`Component getGlassPane()`	It gets the `glassPane` object for this frame
`JMenuBar getJMenuBar()`	It gets the menu bar set on this frame
`JLayeredPane getLayeredPane()`	It gets the `layeredPane` object for this frame
`JRootPane getRootPane()`	It gets the `rootPane` object for this frame
`protected boolean isRootPaneCheckingEnabled()`	It indicates whether calls to `add` and `setLayout` are forwarded to the `contentPane`
`protected String paramString()`	It gets a string representation of this `JFrame`
`protected void processWindowEvent(WindowEvent e)`	It processes window events occurring on this component
`void remove(Component comp)`	It removes the indicated component from this container
`void repaint(long time, int x, int y , int width, int height)`	It repaints the specified rectangle of this component within `time` milliseconds
`void setContentPane (Container contentPane)`	It sets the `contentPane` property
`void setDefaultCloseOperation (int operation)`	It sets the operation that will happen by default when the user closes this frame

Table 11.17: Methods of the JFrame class	
Method	**Does this**
`static void setDefaultLookAndFeelDecorated(boolean defaultLookAndFeelDecorated)`	It sets whether the newly created `JFrames` should have their window decorations provided by the current look and feel
`void setGlassPane(Component glassPane)`	It sets the `glassPane` property
`void setIconImage(Image image)`	It sets the image to be displayed as the icon for the window
`void setJMenuBar (JMenuBar menubar)`	It sets the menu bar for this frame
`void setLayeredPane (JLayeredPane layeredPane)`	It sets the `layeredPane` property
`void setLayout (LayoutManager manager)`	It sets the `LayoutManager`
`protected void setRootPane (JRootPane root)`	It sets the `rootPane` property
`protected void setRootPaneCheckingEnabled (boolean enabled)`	It determines whether calls to `add()` and `setLayout()` are forwarded to the `contentPane`
`void setTransferHandler (TransferHandler newHandler)`	It sets the `transferHandler` property
`void update(Graphics g)`	It calls the `paint(g)` method

We'll put the `JFrame` class to work in an application now. In this case, we'll create a `JPanel` object (as in the previous solution), install the panel in a `JFrame` object, and display the text "Kogent Learning Solutions!" in the panel. We will set the background color of the panel as pink. The code is as follows:

```
class TextPanel extends JPanel {
TextPanel() { setBackground(Color.pink); }
public void paintComponent (Graphics g) {
super.paintComponent(g);
g.drawString("Kogent Learning Solutions!", 50, 80); }
}
```

Next, we create an object of this new class and add it to the content pane of the `JFrame` object in an application, as shown in the following code:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class SwingAppDemo extends JFrame {
TextPanel panel;
public SwingAppDemo() {
super("Swing Application Demo");
Container c = getContentPane();
panel = new TextPanel();
c.add(panel); }
public static void main(String args[]) {
final JFrame frame = new SwingAppDemo();
frame.setBounds(100, 100, 250, 200);
frame.setVisible(true); }
}
```

In addition, note that we have used the `setBounds()` method to set the screen location and size of the `JFrame` window. The result appears as shown in Figure 11.7:



Figure 11.7: Displaying a Swing Application in a Frame Window

However, there's a problem with the application you see in Figure 11.7. When you close the application window by clicking the close button, it disappears from the screen (which is more than an AWT application window would do), but the application does not end. To remedy this situation, take a look at the next solution.

Closing **JFrame** Windows

"Hey," says the NP, "Java's gone all wacky again. When I try to close a Swing `JFrame` window, nothing happens." "Well," you say, "you can take care of that in several ways." "I'll get some coffee," the NP says.

When you close an AWT window, nothing happens by default, but you can handle window events yourself and end the application if you like. Swing `JFrame` windows, on the other hand, let you set a default close operation with the `setDefaultCloseOperation()` method. The possible values that you can pass to this method are as follows:

- `DO_NOTHING_ON_CLOSE`—Nothing happens when the window is tried to close.
- `HIDE_ON_CLOSE`—It hides the window when it's closed.
- `DISPOSE_ON_CLOSE`—It hides and disposes the window when it's closed. In addition, any resources used by the window also get freed. You cannot redisplay the window although the window object is still available in the memory.
- `EXIT_ON_CLOSE`—It exits the application on close of the window.

NOTE

The EXIT_ON_CLOSE actually calls the System.exit(0) method and therefore, abruptly terminates the application or program. The DISPOSE_ON_CLOSE, however, hides and disposes only the Jframe, not the other independent parts of the program; they still remain active.

Note that these possibilities only deal with the window itself; if you want to end the application when the application window is closed, you'll still have to handle that yourself. Here, we dispose of the window and end the application with a window adapter inner class, as shown in the following code:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class JFrameCloseDemo extends JFrame {
        TextPanel panel;
        public JFrameCloseDemo() {
                super("JFrame Close Demo");
                Container c = getContentPane();
                panel = new TextPanel();
                c.add(panel); }
        public static void main(String args[]) {
                final JFrame frame1 = new JFrameCloseDemo();
                frame1.setBounds(100, 100, 300, 200);
                frame1.setVisible(true);
                frame1.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
                frame1.addWindowListener(new WindowAdapter() {
                public void windowClosed(WindowEvent e){
                System.exit(0); }});
        }
```

```
    }
    class TextPanel extends JPanel {
    TextPanel() { setBackground(Color.cyan); }
      public void paintComponent (Graphics g) {
            super.paintComponent(g);
            g.drawString("Swing application using JPanel and JFrame!", 30, 60);
            g.drawString("Close me to end the application!", 35, 100); }
    }
```

Figure 11.8 shows the result of the program `JFrameCloseDemo.java`:

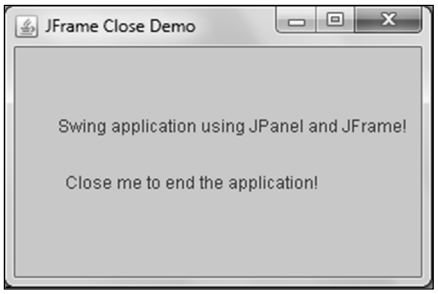


Figure 11.8: Closing the Application As Well

The `main()` method is always a static method, but Swing frames are objects. So, how does the program get executed? This can be answered by simply creating the application as a class that extends `JFrame`. The `main` method just calls the class constructor, which performs the real work of the application.

Alternatively, you can pass `EXIT_ON_CLOSE` to `setDefaultCloseOperation`. `EXIT_ON_CLOSE` causes the program to exit when the window is closed. Using `EXIT_ON_CLOSE` means that the program will exit when the window is closed, and you can omit the inner window adapter class code.

Let's consider a simple program that uses `EXIT_ON_CLOSE`. This frame is defined by the `JFrame` class. The program just displays the text "Welcome to Swing Programming!" on the title bar, as shown in the following code:

```
    import javax.swing.*;
    import java.awt.*;
    import java.lang.*;
    public class Welcome extends JFrame {
    public static void main(String[] args) { new Welcome(); }
      public Welcome() {
            this.setSize(400,200);
            this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            this.setTitle("Welcome to Swing Programming!");
            this.setVisible(true); }
    }
```

The result of the code is shown in Figure 11.9:



Figure 11.9: JFrame Displaying Text on the Title Bar

Let's take a look at the following program that uses `EXIT_ON_CLOSE`:

```
    import javax.swing.JFrame;
    import java.awt.Toolkit;
```

```
import java.awt.*;
public class ExitDemo {
   static JFrame jframe1 = new JFrame("The Frame Window being Sized");
   public static void main(String[] args) {
         Toolkit kit = jframe1.getToolkit();
             Dimension d =  kit.getScreenSize();
         int x = d.width/3;
         int y = (d.height)/3 - jframe1.getHeight();
         jframe1.setBounds(x, y, d.width/3 , d.height/3);
         jframe1.setVisible(true);
         jframe1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); }
}
```

In the preceding code, a `Toolkit` object is first created and used for getting the dimensions of the screen. The x position is set to the horizontal center of the screen after dividing the screen width by 3. Next, the y position is set to the vertical center (by dividing the screen height by 3), which is less than the width of the frame. The `setBounds()` sets the position and size of the frame. The `setDefaultCloseOperation (int operation) void()` sets the close operation to `JFrame.EXIT_ON_CLOSE` so that the frame is closed when the program terminates.

Using Insets for Selecting Component Borders

"Say," says the NP, "the Big Boss wants our programs to look more decorative, more visually interesting. Any ideas?" "Well," you say, "you can use different kinds of borders for components in Swing. That's easy enough." "How does it work?" the NP asks, enthralled.

To create a border for a component, you can use the `BorderFactory` class, which can create borders of these classes: `BevelBorder`, `SoftBevelBorder`, `EtchedBorder`, `LineBorder`, `TitledBorder`, and `MatteBorder`. You can also use the `EmptyBorder`, `CompoundBorder`, and `AbstractBorder` classes to create your own borders. The inheritance diagram for the `BorderFactory` class is as follows:

```
java.lang.Object
      |____javax.swing.BorderFactory
```

You'll find the `BorderFactory` class methods in Table 11.18:

Table 11.18: Methods of the BorderFactory class

Methods	Does this
`static Border createBevelBorder (int type)`	It creates a beveled border of the indicated type using brighter shades of the component's current background color for highlighting and darker shading for shadows
`static Border createBevelBorder (int type, Color highlight, Color shadow)`	It creates a beveled border of the indicated type with the indicated highlighting and shadowing
`static Border createBevelBorder (int type, Color highlightOuter, Color highlightInner, Color shadowOuter, Color shadowInner)`	It creates a beveled border of the indicated type using the indicated colors for the inner and outer highlight and shadow areas
`static CompoundBorder createCompoundBorder()`	It creates a compound border with a null inside edge and a null outside edge
`static CompoundBorder createCompoundBorder (Border outsideBorder, Border insideBorder)`	It creates a compound border, indicating the border objects for the outside and inside edges
`static Border createDashedBorder(Paint paint)`	It creates a dashed border of the indicated `paint`
`static Border createDashedBorder(Paint paint, float length, float spacing)`	It creates a dashed border of the indicated `paint`, relative `length`, and `spacing`

Table 11.18: Methods of the BorderFactory class

Methods	Does this
`static Border createDashedBorder(Paint paint, float thickness, float length, float spacing, Boolean rounded)`	It creates a dashed border of the indicated `paint`, `thickness`, line shape, relative `length`, and `spacing`
`static Border createEmptyBorder()`	It returns an empty border that takes up no space
`static Border createEmptyBorder(int top, int left, int bottom, int right)`	It returns an empty border that takes up space but does no drawing, specifying the width of the top, left, bottom, and right sides
`static Border createEtchedBorder()`	It creates a border with an etched look using the component's current background color for highlighting and shading
`static Border createEtchedBorder(Color highlight, Color shadow)`	It creates a border with an etched look using the indicated highlighting and shading colors
`static Border createEtchedBorder(int type)`	It creates a border with an etched look using the component's current background color for highlighting and shading
`static Border createEtchedBorder(int type, Color highlight, Color shadow)`	It creates a border with an etched look using the indicated highlighting and shading colors
`static Border createLineBorder(Color color)`	It creates a line border with the indicated color
`static Border createLineBorder(Color color, int thickness)`	It creates a line border with the indicated color and width
`static Border createLoweredBevelBorder()`	It creates a beveled border with a lowered edge and softened corners using brighter shades of the component's current background color for highlighting and darker shading for shadows
`static MatteBorder createMatteBorder(int top, int left, int bottom, int right, Color color)`	It creates a matte-look border using a solid color
`static MatteBorder createMatteBorder(int top, int left, int bottom, int right, Icon tileIcon)`	It creates a matte border made up of multiple tiles of the indicated icon
`static Border createRaisedBevelBorder()`	It creates a beveled border with a raised beveled edge using brighter shades of the component's current background color for highlighting and darker shading for shadows
`static Border createRaisedSoftBevelBorder()`	It creates a beveled border with a raised beveled edge and softened corners using brighter shades of the component's current background color for highlighting and darker shading for shadows
`static Border createSoftBevelBorder (int type)`	It creates a beveled border of the indicated type with softened corners using brighter shades of the component's current background color for highlighting and darker shading for shadows
`static Border createSoftBevelBorder(int type, Color highlight, Color shadow)`	It creates a beveled border of the indicated type with softened corners using the indicated highlighting and shadowing
`static Border createSoftBevelBorder (int type, Color highlightOuter, Color highlightInner, Color shadowOuter, Color shadowInner)`	It creates a beveled border of the indicated type with softened corners using the indicated colors for the inner and outer edges of the highlight and shadow areas
`static Border createStrokeBorder (BasicStroke stroke)`	It creates a border of the specified `stroke`

Table 11.18: Methods of the BorderFactory class	
Methods	**Does this**
`static Border createStrokeBorder (BasicStroke stroke, Paint paint)`	It creates a border of the specified `stroke` and `paint`
`static TitledBorder createTitledBorder(Border border)`	It creates a new titled border with an empty title, the indicated border object, the default text position, the default justification, and the default font and text color
`static TitledBorder createTitledBorder(Border border, String title)`	It adds a title to an existing border with default positioning, default justification, and default font and text color
`static TitledBorder createTitledBorder(Border border, String title, int itleJustification, int titlePosition)`	It adds a title to an existing border with the indicated positioning and default font and text color
`static TitledBorder createTitledBorder(Border border, String title, int itleJustification, inttitlePosition, Font titleFont)`	It adds a title to an existing border with the indicated positioning and using the default font and text color
`static TitledBorder createTitledBorder(Border border, String title, int itleJustification, inttitlePosition, Font titleFont, Color titleColor)`	It adds a title to a border with the indicated positioning, font, and color
`static TitledBorder createTitledBorder(String title)`	It creates a new titled border with the indicated title, default border type, default text position, default justification, and default font and text color

The `BorderFactory` class creates objects that implement the `Border` interface; you'll find the methods of this interface in Table 11.19:

Table 11.19: Methods of the Border interface	
Methods	**Does this**
`Insets getBorderInsets (Component c)`	It gets the insets of the border
`boolean isBorderOpaque()`	It determines whether the border is opaque
`void paintBorder(Component c, Graphics g, int x, int y, int width, int height)`	It paints the border for the indicated component with the indicated position and size

Using Insets

An important aspect of working with borders is to know about *insets*, which indicate the distance you must allow along each edge to account for the border. You can get the insets of a border with the `getBorderInsets()` method of the `Border` interface; this method returns an object of the class `Insets`. The inheritance diagram for the `Insets` class is given as follows:

```
java.lang.Object
  |____java.awt.Insets
```

You'll find the fields of the `Insets` class in Table 11.20, its constructor in Table 11.21, and its methods in Table 11.22:

Table 11.20: Fields of the Insets class	
Field	**Does this**
`int bottom`	The inset from the bottom
`int left`	The inset from the left

Table 11.20: Fields of the Insets class	
Field	**Does this**
`int right`	The inset from the right
`int top`	The inset from the top

Table 11.21: Constructor of the Insets class	
Constructor	**Does this**
`Insets(int top, int left, int bottom, int right)`	It creates and initializes a new `Insets` object with the specified top, left, bottom, and right insets

Table 11.22: Methods of the Insets class	
Method	**Does this**
`Object clone()`	It creates a copy of this object
`boolean equals(Object obj)`	It checks whether two insets objects are equal
`int hashCode()`	It returns the hashcode for this insets
`void set(int top, int left, int bottom, int right)`	It sets the top, left, bottom, and right to the given values
`String toString()`	It gets a string representing this `Insets` object

Let's consider an example in which we add a border to the `JPanel` object used in the applet. To do that, we just use the panel's `setBorder()` method (which most visual components have in Swing) and create a new titled border with the `BorderFactory` class's `createTitledBorder()` method, as shown in the following code:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
/* <APPLET
  CODE=BorderDemo.class
  WIDTH=300
  HEIGHT=200 >
</APPLET>*/
public class BorderDemo extends JApplet {
        TextPanel panel;
        public void init() {
        Container c = getContentPane();
        panel = new TextPanel();
        panel.setBorder(BorderFactory.createTitledBorder("Text Panel"));
        c.add(panel); }
}
```

Besides adding a new border to the panel, we take the insets of this border into account to draw the message "Swing with Border Insets!" without overlapping the border. To do that, we use the `getBorder()` method to get the current border, the `getBorderInsets()` method of the object that supports the `Border` interface, and the left data member of that object to determine where to start printing the message so as not to overlap the border, as shown in the following code:

```
class TextPanel extends JPanel {
        public void paintComponent (Graphics g) {
                g.drawString("Swing with Border Insets!",
                getBorder().getBorderInsets(this).left, 100); }
  }
```

The result appears as shown in Figure 11.10:

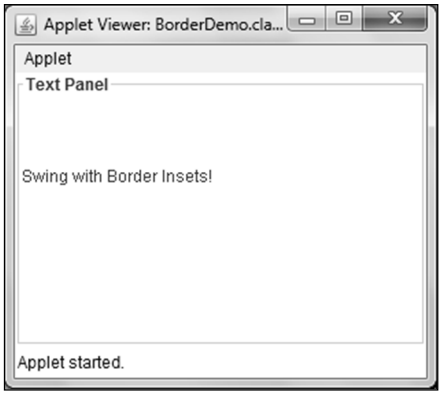


Figure 11.10: Adding a Border to a JPanel Object

You can see in this figure that the text appears inside the thin titled border that encloses the panel.

The Synth Skinnable Look and Feel

A customizable look and feel was introduced to project Swing in Java SE 5, which is well known as Synth, where the UI known as its "skin" is controlled by an XML file. It is possible to create a custom look without writing code by simply loading an XML file with the component definitions rather than customizing the look and feel by providing default properties to the `UIManager` in a Properties table. To demonstrate this, here's an example of using the Synth look and feel:

```
    import java.awt.BorderLayout;
    import java.awt.EventQueue;
    import java.io.InputStream;
    import java.text.ParseException;
    import javax.swing.JButton;
    import javax.swing.JLabel;
    import javax.swing.JFrame;
    import javax.swing.JFrame.*;
    import javax.swing.UIManager;
    import javax.swing.plaf.synth.SynthLookAndFeel;
    public class SynthDemo {
      public static void main(String args[]) {
            Runnable r = new Runnable() {
                    public void run() {
                            SynthLookAndFeel synth = new SynthLookAndFeel();
                    try {
                            Class c = SynthDemo.class;
                            InputStream is =
                            c.getResourceAsStream("synth.xml");
                            if (is == null) {
                             System.err.println("Missing configuration file");
                             System.exit(-1); }
                            synth.load(is, c); }
                    catch (ParseException e) {
                    System.err.println("Bad configuration file");
                    System.exit(-2); }
            try { UIManager.setLookAndFeel(synth); }
            catch (javax.swing.UnsupportedLookAndFeelException e) {
                    System.err.println("Cannot change to Synth");
                    System.exit(-3); }
            JFrame jframe1 = new JFrame("Synth Look and Feel");
```

```
            jframe1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            JLabel jlabel1 = new JLabel("Synth look and feel");
            jlabel1.setHorizontalAlignment(JLabel.CENTER);
            jframe1.add(jlabel1);
            jframe1.setSize(300,100);
            jframe1.setVisible(true); }
      };
      EventQueue.invokeLater(r); }
      }
```

The `SynthDemo` program creates a new `SynthLookAndFeel` object and loads an XML file, `synth.xml`, with `synth.load` (`InputStream` class). This means that you need the XML file to complete this first example, as shown in the following code:

```
<synth>
   <style id="label">
         <font name="Monospaced" size="18" style="BOLD ITALIC"/>
         <opaque value="true"/>
         <state>
         <color value="RED" type="FOREGROUND"/>
         </state>
   </style>
   <bind style="label" type="region" key="Label"/>
</synth>
```

Run the `SynthDemo` program with the `synth.xml` file. The program reads in the XML file, creates a `JLabel` with a font `"Monospaced"`. The `size` and `style` of the text of the label are `18` and `BOLD ITALIC`, respectively. This label is then added on the screen, as shown in Figure 11.11:

```
C:\>java SynthDemo synth.xml
```

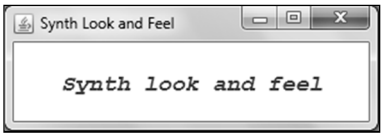


Figure 11.11: Showing the Synth Look and Feel Window

Note the absence of any color or font configuring of the label in the program. Instead, that's done through the XML file. The `synth.xml` file configures the label as follows: 18-point bold-italic font, opaque, and foreground color of red (text color). The `SynthDemo` program then draws the label according to these specifications.

If the preceding font–color combination does not seem to be very appealing, it is possible to change the color, font, and background by making the changes in the XML file depending on the extent to which you can make it more appealing. The font can be made bold if not italic, the color can be changed to any—red, blue, black, and the foreground can be changed to the desired color combination, as shown in the following code:

```
<synth>
   <style id="label">
         <opaque value="true"/>
         <font name="Monospaced" size="21" style="BOLD"/>
         <state>
         <color value="BLUE" type="FOREGROUND"/>
         </state>
   </style>
   <bind style="label" type="region" key="Label"/>
</synth>
```

Therefore, the complete look of a program can be changed by simply changing the name of the XML file and recompiling the program. In the above code, we set the `font`, `size`, and `style` of the label as `"Monospaced"`, `28`, and `BOLD`, respectively. This label is then added on the screen, as shown in Figure 11.12:

```
C:\>java SynthDemo1 synth1.xml
```



Figure 11.12: Showing the Synth Look and Feel Window with Changed XML File Name

Here, we make a new program `SynthDemo1.java` calling the XML file `synth1.xml`.

This shows the capability of the Synth look and feel. Without any reprogramming, you can change the look and feel of the program. A Synth File Format document contains the file format used in the XML file. You can see in `synth.xml` that the outermost tag is `<synth>`. It can have other tags, such as `<style>`, `<bind>`, `<font>`, `<color>`, `<imagePainter>`, and `<imageIcon>` inside it. In addition, there are several other special properties that you can set. These properties are enlisted in a Component Specific Properties document. As seen in the just stated preceding example, you define your styles and then bind them to the components. Most of the information is contained in the `<style>` element. The other elements that can be configured for each named style are:

```
<property>
<defaultsProperty>
<state>
<font>
<painter>
<imagePainter>
<backgroundImage>
<opaque>
<imageIcon>
<inputMap>
```

You can specify the font's name, size, and style using the `<font>` tag. The other styles supported are `PLAIN`, `BOLD`, and `ITALIC`. To specify two styles at the same time, both are written inside quotes with a space between the two:

```
<font style="BOLD ITALIC"/>
```

However, the `<state>` tag is not always simple as it can be seen in the earlier examples. It can be more complex too. For example, you can have a different look of a button when the mouse moves it. In such a case, you can configure the `MOUSE_OVER` state as follows:

```
<state value="MOUSE_OVER">
  <font name="Serif" size="20" style="BOLD"/>
</state>
```

This makes the appropriate state change to 20 point, bold, and `Serif` font. Besides the `MOUSE_OVER state`, the other mouse state settings available are:

- ENABLED
- PRESSED
- DISABLED
- FOCUSED
- SELECTED
- DEFAULT

Depending upon the component, you can change the properties of any or all of these states. But, it must be ensured that the size of the component remains unchanged with change in the configuration. Else, the component will not revalidate itself when the state changes. In addition, the `<imageIcon>` element provides you the means to work with icons. For example, you can have the following style to change the default tree icons:

```
<style id="tree">
<imageIcon id="collapse" path="collapse.jpg"/>
<property key="Tree.collapsedIcon" value="collapse"/>
```

```
    <imageIcon id="expand" path="expand.jpg"/>
    <property key="Tree.expandedIcon" value="expanded"/>
    </style>
```

Note that you must bind the `<style>` element, used in the previous example, to the tree component. The image files `collapse.jpg` and `expand.jpg` are then located relative to the `Class` object and passed into the method `load` to create the `Synth` look and feel.

Synth also supports long-term persistence of Java Beans components to embed object references in the configuration file. Generally, we embed the `Painter` object reference in it. For example, the following element is written in the configuration file:

```
    <object id="grad" class="GradientPaint"/>
```

The preceding element signifies a class named `GradientPaint` identified by the gradient id `grad`. This class extends the `SynthPainter` class. Moreover, if the file contains the following element:

```
    <painter method="textFieldBackground" idref="grad"/>
```

it signifies that the `paintTextFieldBackground()` method of the `GradientPainter` class for which the `idref` of painter matches with the `id` of object is called for the component with which this painter is associated.

Unfortunately, you can't customize the feel of a component using `Synth`; for example, you can't have a customized way (such as voice activation) to click a button.

To conclude, separating the logic of an application from its look considerably helps in making its maintenance easier. Whether you create a new application or use an existing one, you may switch to the Synth look and feel when appropriate.

The Pluggable Look and Feel

The Big Boss is back and says, "I understand you can set the appearance of a Swing program to match various programming platforms." "That's right," you say. The Big Boss asks, "Do you have anything in burnished mahogany?" "Sorry," you reply.

By default, the appearance of Swing programs is set to the Java Metal look, a cross-platform look that Sun has designed. However, you can change that by using the `UIManager` class. The inheritance diagram for this class is as follows:

```
    java.lang.Object
      |____javax.swing.UIManager
```

You'll find the constructor of the `UIManager` class in Table 11.23 and its methods in Table 11.24:

Table 11.23: The constructor of the UIManager Class

Constructor	Does this
`UIManager()`	It constructs a new `UIManager` object.

Table 11.24: Methods of the UIManager Class

Method	Does this
`static void addAuxiliaryLookAndFeel (LookAndFeel laf)`	It adds a `LookAndFeel` object to the list of auxiliary look and feels.
`static void addPropertyChangeListener (PropertyChangeListener listener)`	It adds a `PropertyChangeListener` object.
`static Object get (Object key)`	It gets an object from the defaults table.
`static Object get (Object key, Locale l)`	It gets an object from the defaults table that is appropriate for the given locale.
`static LookAndFeel[] getAuxiliaryLookAndFeels()`	It gets the list of auxiliary look and feel types.

Table 11.24: Methods of the UIManager Class

Method	Does this
`static boolean getBoolean(Object key)`	It gets a boolean from the defaults table which is associated with the given key value.
`static boolean getBoolean(Object key, Locale l)`	It gets a boolean from the defaults table which is associated with the given key value and locale.
`static Border getBorder(Object key)`	It gets a border from the defaults table which is associated with the given key value.
`static Border getBorder(Object key, Locale l)`	It gets a border from the defaults table that is appropriate for the given locale.
`static Color getColor(Object key)`	It gets a drawing color from the defaults table.
`static Color getColor(Object key, Locale l)`	It gets a drawing color from the defaults table that is appropriate for the given locale.
`static String getCrossPlatformLookAndFeelClassName()`	It gets the name of the `LookAndFeel` class that implements the default cross-platform look and feel (Java look and feel).
`static UIDefaults getDefaults()`	It gets the defaults for this look and feel.
`static Dimension getDimension(Object key)`	It gets a dimension from the defaults.
`static Font getFont (Object key)`	It gets a font from the defaults.
`static Font getFont (Object key, Locale l)`	It gets a font from the defaults, which is appropriate for the given locale.
`static Icon getIcon (Object key)`	It gets an icon from the defaults, which is appropriate for the given locale.
`static Icon getIcon (Object key, Locale l)`	It gets an icon from the defaults.
`static Insets getInsets (Object key)`	It gets an `Insets` object from the defaults.
`static Insets getInsets (Object key, Locale l)`	It gets an `Insets` object from the defaults that is appropriate for the given locale.
`static UIManager.LookAndFeelInfo[] getInstalledLookAndFeels()`	It gets an array of `LookAndFeelInfo` objects that hold information about the `LookAndFeel` implementations currently installed.
`static int getInt (Object key)`	It gets an integer from the defaults .
`static int getInt (Object key, Locale l)`	It gets an integer from the defaults, which is appropriate for the given locale.
`static LookAndFeel getLookAndFeel()`	It gets the current default look and feel.
`static UIDefaults getLookAndFeelDefaults()`	It gets the default values for this look and feel or null.

Table 11.24: Methods of the UIManager Class

Method	Does this
`static PropertyChangeListener[] getPropertyChangeListeners ()`	It gets an array of `PropertyChangeListeners` added to this UIManager.
`static String getString(Object key)`	It gets a string from the defaults.
`static String getString(Object key, Locale l)`	It gets a string from the defaults, which is appropriate for the given locale.
`static String getSystemLookAndFeelClassName()`	It gets the name of the `LookAndFeel` class that implements the native systems look and feel if it exists; otherwise, name of the default `LookAndFeel` class.
`static ComponentUI getUI(JComponent target)`	It gets the appropriate `ComponentUI` implementation for `target`.
`static void installLookAndFeel(String name, String className)`	It adds the indicated look and feel to the set of available look and feels.
`static void installLookAndFeel(UIManager.LookAndFeelInfo info)`	It adds the indicated look and feel to the set of available look and feels.
`static Object put(Object key, Object value)`	It stores an object in the defaults table.
`static boolean removeAuxiliaryLookAndFeel (LookAndFeel laf)`	It removes a `LookAndFeel` object from the list of auxiliary look and feel types.
`static void removePropertyChangeListener (PropertyChangeListener listener)`	It removes a `PropertyChangeListener` object from the listener list.
`static void setInstalledLookAndFeels (UIManager.LookAndFeelInfo [] infos)`	It sets the set of currently installed look and feels.
`static void setLookAndFeel (LookAndFeel newLookAndFeel)`	It sets the current look and feel to the `newLookAndFeel` object.
`static void setLookAndFeel (String className)`	It loads the current look and feel with a given class name and passes it to the `setLookAndFeel(LookAndFeel)` method.

To change the look and feel as a program is tuning, you can use the `UIManager` class's `setLookAndFeel()` method, passing it one of these arguments as a string:

- **javax.swing.plaf.metal.MetalLookAndFeel—** Pass this argument for the Metal look and feel.
- **com.sun.java.swing.plaf.motif.MotifLookAndFeel—** Pass this argument for the Motif look and feel.
- **com.sun.java.swing.plaf.windows.WindowsLookAndFeel—** Pass this argument for the Windows look and feel.

After changing the look and feel of the content pane, you make it take effect with the `updateComponentTreeUI()` method of the `SwingUtilities` class like this:

```
SwingUtilities.updateComponentTreeUI(getContentPane());
```

Here's an example in which we add a Swing button and three radio buttons to see how the change in look and feel changes the appearance of controls. The three radio buttons in this applet are labeled `MetalLook`, `MotifLook`, and `WindowsLook`. When you click a radio button, the applet will change to the corresponding look and feel. The code is as follows:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.plaf.metal.MetalLookAndFeel;
import com.sun.java.swing.plaf.motif.MotifLookAndFeel;
import com.sun.java.swing.plaf.windows.WindowsLookAndFeel;
/* <APPLET
  CODE=LookAndFeelDemo.class
  WIDTH=300
  HEIGHT=200>
</APPLET> */
public class LookAndFeelDemo extends JApplet implements ActionListener{
  JPanel jpanel1;
  JRadioButton jradio1, jradio2, jradio3;
  ButtonGroup group;
  JButton jbutton1;
public void init() {
jpanel1 = new JPanel();
group = new ButtonGroup();
jbutton1 = new JButton("Kogent");
jradio1 = new JRadioButton("MotifLook");
group.add(jradio1);
jradio1.addActionListener(this);
jradio2 = new JRadioButton("MetalLook");
group.add(jradio2);
jradio2.addActionListener(this);
jradio3 = new JRadioButton("WindowsLook");
group.add(jradio3);
jradio3.addActionListener(this);
jpanel1.add(jradio1);
jpanel1.add(jradio2);
jpanel1.add(jradio3);
jpanel1.add(jbutton1);
Container c = getContentPane();
c.add(jpanel1);  }
public void actionPerformed(ActionEvent e) {
  JRadioButton src = (JRadioButton)e.getSource();
        try {
                if((JRadioButton)e.getSource() == jradio2)
                 UIManager.setLookAndFeel(
                 "javax.swing.plaf.metal.MetalLookAndFeel");
                else if((JRadioButton)e.getSource() == jradio1)
                UIManager.setLookAndFeel(
                  "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
                else if((JRadioButton)e.getSource() == jradio3)
                  UIManager.setLookAndFeel(
                  "com.sun.java.swing.plaf.windows.WindowsLookAndFeel"); }
                catch(Exception ex) { }
                SwingUtilities.updateComponentTreeUI(getContentPane()); }
}
```

You can see the applet in the Metal look and feel in Figure 11.13, the Motif look and feel in Figure 11.14, and the Windows look and feel in Figure 11.15:



Figure 11.13: The Metal Look and Feel

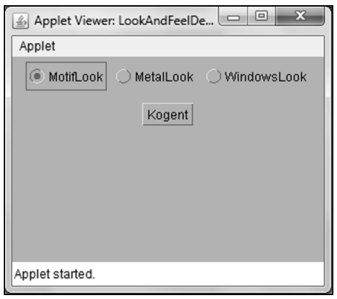


Figure 11.14: The Motif Look and Feel

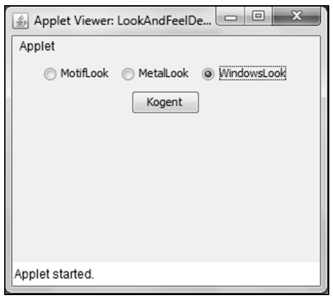


Figure 11.15: The Windows Look and Feel

Switching between these looks is as easy as clicking a radio button.

The Pluggable Look and Feel for Components

"Good lord!," says the NP, "Wouldn't it be great if all components used the same fonts and borders?" "Well," you say, "I'm not sure about that, but you can actually make that happen in Swing." The NP asks, "You can?"

You can set the look and feel of individual components in Swing using the `LookAndFeel` class. The inheritance diagram for that class is as follows:

```
java.lang.Object
  |____javax.swing.LookAndFeel
```

You'll find the constructor for the `LookAndFeel` class in Table 11.25 and its methods in Table 11.26:

Table 11.25: The constructor of the LookAndFeel Class

Constructor	Does this
`LookAndFeel()`	It creates a `LookAndFeel` object.

Table 11.26: Methods of the LookAndFeel Class

Method	Does this
`UIDefaults getDefaults()`	It gets the default look and feel..
`abstract String getDescription()`	It gets a one-line description of this look and feel implementation.
`static Object getDesktopPropertyValue (String systemPropertyName, Object fallbackValue)`	It gets the value of the specified system desktop property by invoking `Toolkit.getDefaultToolkit().getDesktopProperty()`.
`Icon getDisabledIcon(JComponent component, Icon icon)`	It gets an `Icon` with the disabled appearance.
`Icon getDisabledSelectedIcon(JComponent component, Icon icon)`	It gets an `Icon` to be used by disabled components that are also selected.
`abstract String getID()`	It gets a string that identifies this look and feel.
`LayoutStyle getLayoutStyle()`	It returns the `LayoutStyle` object for this look and feel.
`abstract String getName()`	It gets a short string that identifies this look and feel.

Table 11.26: Methods of the LookAndFeel Class

Method	Does this
`boolean getSupportsWindowDecorations()`	It returns true if the LookAndFeel returned RootPaneUI instances support providing Window decorations in a JRootPane. The default implementation returns false; subclasses that support Window decorations should override this and return true.
`void initialize()`	It initializes the look and feel.
`static void installBorder (JComponent c, String defaultBorderName)`	It installs a component's default `Border` object.
`static void installColors (JComponent c, String defaultBgName, String defaultFgName)`	It installs a component's foreground and background color properties with values from the current defaults.
`static void installColorsAndFont (JComponent c, String defaultBgName, String defaultFgName, String defaultFontName)`	It installs a component's foreground, background, and font properties with values from the defaults.
`static void installProperty (JComponent c, String propertyName, Object propertyValue)`	It installs a property with specified name and value on a component if that property does not already exist.
`abstract boolean isNativeLookAndFeel()`	If True, this is an implementation of the underlying platform's "native" look and feel.
`abstract boolean isSupportedLookAndFeel()`	If True, the underlying platform supports and/or permits this look and feel.
`static void loadKeyBindings(InputMap retMap, Object[] keys)`	It populates an `InputMap` object with the specified bindings.
`static ComponentInputMap makeComponentInputMap (JComponent c, Object[] keys)`	It creates a `ComponentInputMapUIResource` resource from `keys`.
`static Object makeIcon (Class<?> baseClass, String gifFile)`	It creates a `UIDefaults.LazyValue` that creates an `ImageIcon` for the specified `gifFile` file name.
`static InputMap makeInputMap(Object[] keys)`	It creates an `InputMapUIResource` from the `keys`.
`static JTextComponent.KeyBinding[] makeKeyBindings(Object[] keyBindingList)`	It builds an array of `KeyBindings`.
`void provideErrorFeedback (Component component)`	It is invoked when the user attempts an invalid operation (such as pasting into an uneditable `JTextField`)
`string toString()`	It gets a string that identifies this object's properties.
`void uninitialize()`	It uninitializes the look and feel.
`static void uninstallBorder (JComponent c)`	It uninstalls a component's default border if the border is an instance of `UIResource`.

Here's an example in which we give a Swing text field the look and feel of a Swing label using the `LookAndFeel` class's `installBorder()` and `installColorsAndFont()` methods. Note that you have not yet seen these controls at work formally; this example is just to indicate what can be done with the look and feel types. We'll create a Swing label here and a class extended from the text field class to look like a label. The code is as follows:

```
import java.awt.*;
import javax.swing.*;
/* <APPLET
   CODE=LookAndFeelDemo1.class
   WIDTH=200
   HEIGHT=200 >
</APPLET> */
public class LookAndFeelDemo1 extends JApplet {
public void init() {
Container c = getContentPane();
TestJTextField test = new TestJTextField("This is a text field.");
c.setLayout(new FlowLayout());
c.add(new JLabel("This is a swing label."));
c.add(test); }
}
class TestJTextField extends JTextField {
public  TestJTextField (String str) { super(str); }
public void updateUI() {
super.updateUI();
setHighlighter(null);
setEditable(false);
LookAndFeel.installBorder(this, "Label.border");
LookAndFeel.installColorsAndFont(this, "Label.background",
"Label.foreground", "Label.font"); }
   }
```

The result appears as shown in Figure 11.16:



Figure 11.16: Setting the Look and Feel of a Text Field to be Like a Label

You can see in the figure that the text field, which appears under the label, has been given the look and feel of the label. This gives you an indication of how powerful the look and feel handling in Swing is.

The Nimbus Look and Feel

'I've heard someone saying Java also possesses Nimbus look and feel, is it true?," asks the NP. "Yes dear", you say, "Java SE 6 Update 10 introduced the Nimbus look and feel." The NP says, "I want to know more about it." You say, "Ok."

After having created a cross-platform look and feel in the form of Metal, Sun enhanced its Swing capabilities further, and added a new and different cross-platform look known as Nimbus look and feel. As stated earlier also, it debuted in Java SE 6 Update 10. Instead of using static bitmaps, the Nimbus look and feel uses the Java2D vector graphics to draw UI elements. This gives the elements an attractive and elegant look, independent of the resolution. The look can even be customized as per the choice of the user.

In Java SE 7, the class NimbusLookAndFeel is located at the following location: com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel.

To set the look and feel to Nimbus, you call the `UIManager's` method `setLookAndFeel()` and pass the classname as an argument to it as:

```
UIManager.setLookAndFeel("com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel");
```

Here's an example in which we set the look and feel of the Swing components to Nimbus look and feel:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
   CODE=NimbusDemo.class
   WIDTH=300
   HEIGHT=200 >
</APPLET> */
public class NimbusDemo extends JApplet {
public void init() {
Container c = getContentPane();
c.add(new ComponentsPanel(), BorderLayout.CENTER); }
}
class ComponentsPanel extends JPanel {
public ComponentsPanel() {
add(new JButton("Kogent"));
add(new JTextField("Kogent"));
add(new JCheckBox("Kogent"));
add(new JRadioButton("Kogent"));
add(new JLabel("Kogent"));
add(new JList(new String[] {
" Item 1", " Item 2", " Item 3","Item 4","Item 5" }));
add(new JScrollBar(SwingConstants.HORIZONTAL));
try{
UIManager.setLookAndFeel(
"com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"); }
catch(Exception ex) { }
        }
}
```

You can see the applet in the Nimbus look and feel in Figure 11.17:

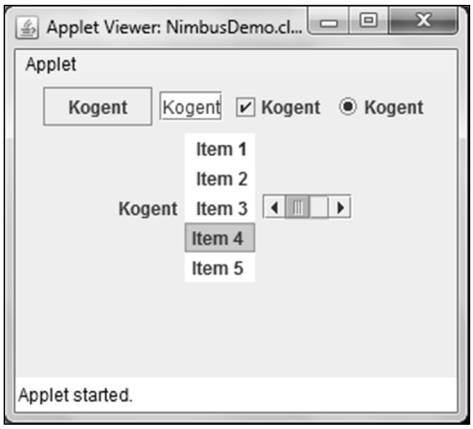


Figure 11.17: The Java Nimbus Look and Feel

Summary

In this chapter, you have learned about the Java foundation classes. Then, you have learned Swings, heavyweight and lightweight components, and features of Swings. The chapter has also explained how to create graphic programming using panes. It has explored the use of Swings and discussed how to create an applet in them. Further, you have learned how to close JFrame windows as well as use of insets to select component borders. Next, synth skinnable look and feel, pluggable look and feel, pluggable look and feel for component, and the nimbus look and feel have been explained.

In the next chapter, you will learn about Swing–Text Fields, Buttons, Toggle buttons, Checkboxes, and Radio buttons.



12

Swing—Text Fields, Buttons, Toggle Buttons, Checkboxes, and Radio Buttons

If you need an immediate solution to:	*See page:*
Using Labels	399
Using Image Icons	401
Using Images in Labels	402
Using Text Fields	403
Setting Text Field Alignment	405
Creating Password Field	406
Creating Text Areas	407
Creating Editor Panes	409
Using HTML in Editor Panes	411
Using RTF Files in Editor Panes	412
Creating Text Panes	412
Setting Text Pane Text Attributes	414
Working with Sound in Applets	416
Working with Sound in Application	417
Abstract Button: The Foundation of Swing Buttons	417
Using Buttons	421

If you need an immediate solution to:	*See page:*
Displaying Images in Buttons	424
Using Rollover and Disabled Images	426
Default Buttons and Mnemonics	427
Using Toggle Buttons	430
Creating Toggle Button Groups	432
Using Checkboxes	433
Using Radio Buttons	436
Using Checkbox and Radio Button Images	440
Getting and Setting the State of Checkboxes and Radio Buttons	441

In Depth

This chapter begins the discussion of Swing controls; specifically, we'll take a look at Swing labels, buttons, text fields, toggle buttons, checkboxes, and radio buttons. All these are essential controls that make up a big part of the foundation of Swing. With the exception of toggle buttons, all these controls should be familiar to you from AWT programming. However, each of these Swing controls has a lot more functionality than the corresponding AWT controls, and we'll emphasize what's new here. (Bear in mind that all Swing controls appear differently in various looks and feels; this is important to remember when you start to program with them.)

Labels and Text Fields

Labels are basic Swing controls that just display one line of text. There was a belief that labels would support multiple lines of text in Swing, but they turned out supporting only one line. On the other hand, they do support something that AWT labels don't—images. You'll see how to use the `ImageIcon` class to add images to labels.

The classes in `javax.swing` package are packed, as shown in Figure 12.1, where the `JComponent` class is the base:

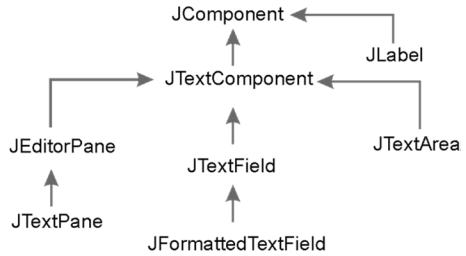


Figure 12.1: The JComponent class tree

The `JLabel` component does not react to input events, so you can't edit it. The `JTextField` component looks similar to a label as it displays a single line of text, but the difference being that it is editable. Moreover, `JFormattedTextField`, a component of `JTextField`, controls and formats the data that you display or enter. The `JEditorPane` and `JTextPane` classes allow you to implement sophisticated editing facilities. On the other hand, the former supports editing of plain text—HyperText Markup Language (HTML) or Rich Text Format (RTF)—the latter enables you to embed images or other components within the text managed by the component.

Text handling in Swing text components has become a somewhat involved solution that we'll look into later in this book. Here, we'll get a start on handling text in Swing with text fields, seeing how they work fundamentally in this chapter and in more depth later.

Password Field

The `JTextComponent` class is extended by two classes: `JTextArea` and `JEditorPane`. `JEditorPane` is extended by `JTextPane` class. `JPasswordField` is a `JTextField` used to write passwords, and the input characters are usually marked with an asterisk. You can also modify the mask character using the `setEchoChar()` method.

Text Area

The `JTextArea` is one of the subclasses of `JTextComponent`. It allows you to edit a multiline text. However, it won't allow you to scroll directly, but you can add scrollbars by adding the text area to the `JScrollPane` container.

Editor Pane

The `JEditorPane` is a special `JTextComponent` that can display and edit HTML 3.2 tags or formats, such as RTF. It provides complex editing facilities in a rather simple and easy way. Notwithstanding, it's just a lightweight HTML viewer generally used to display the help text and not meant to create a full-fledged web

browser. To create an Editor Pane object, either you create it with a `URL` parameter or you use the `setPage()` method to change the web pages. The `HyperlinkListener` interface is also provided to make the links within an HTML page traversable.

Text Pane

The `JTextPane` class, a subclass of `JEditorPane`, lets you associate images or other components with the text managed by the text pane. It supports formatted text, word wrap, and image display like a full-fledged text editor. Additionally, to specify the font formatting it uses a linked list of objects and provides easy to use methods for text formatting.

Text Component Printing

The `javax.swing.text.JTextComponent` class has three print() methods for printing the text component. The `JTextComponent` class is the base class for text components. Each print() method is used to display the print dialog and block until printing is done.

Buttons

In Swing, buttons of all kinds are built on the `AbstractButton` class, and we'll take a look at that class in this chapter. As you might expect, Swing buttons have more capabilities than their AWT counterparts. For example, using Swing buttons, you can display images, use mnemonics (keyboard shortcuts), designate a button as a window's default button, and set a button's margins and text alignment. You can also assign multiple images to a button to handle the case where the mouse moves over the button, and more. You'll see all this in the chapter.

Toggle Buttons

Swing introduces toggle buttons, which are buttons that, when pressed, stay pressed until they are pressed again. Toggle buttons are much like checkboxes and radio buttons that look like standard buttons. In fact, the `JToggleButton` class is the base class for the Swing checkbox and radio button classes, but you can also instantiate objects of this class directly. As with checkboxes and radio buttons, you can also put toggle buttons into groups and use images with them.

Checkboxes and Radio Buttons

The AWT has checkboxes and radio buttons, but handling these controls in Swing is a little different. For one thing, radio buttons have their own class in Swing (radio buttons are just glorified checkboxes in AWT), and you can use images with them. In fact, you'll see that there are a few issues to consider when you use images with checkboxes and radio buttons. That's it for the overview of what's in this chapter. It's time to turn to the "Immediate Solutions" section.

Immediate Solutions

Using Labels

The Novice Programmer appears and says, "I'm creating a new Swing applet, and I want to label the controls in it, so I'm starting with a `Label` **component** and…" "Hold it," you say. "You should be using a `JLabel` component." The NP says, "Oh."

The lightweight Swing label class is `JLabel`. Here's the inheritance diagram for this class:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.JLabel
```

You'll find the constructors for the `JLabel` class in Table 12.1 and its methods in Table 12.2:

Table 12.1: Constructors of the JLabel class

Constructor	Does this
`JLabel()`	It constructs a `JLabel` object with no image and title
`JLabel(Icon image)`	It constructs a `JLabel` object with the indicated image
`JLabel(Icon image, int horizontalAlignment)`	It constructs a `JLabel` object with the indicated image and horizontal alignment
`JLabel(String text)`	It constructs a `JLabel` object with the indicated text
`JLabel(String text, Icon icon, int horizontalAlignment)`	It constructs a `JLabel` object with the indicated text, image, and horizontal alignment
`JLabel(String text, int horizontalAlignment)`	It constructs a `JLabel` object with the indicated text and horizontal alignment

Table 12.2: Methods of the JLabel class

Method	Does this
`protected int checkHorizontalKey (int key, String message)`	It verifies a legal value for the `horizontalAlignment` properties
`protected int checkVerticalKey(int key, String message)`	It verifies a legal value for the `verticalAlignment` or `verticalTextPosition` properties
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` for `JLabel`
`Icon getDisabledIcon()`	It gets the value of the `disabledIcon` property if it has been set
`int getDisplayedMnemonic()`	It gets the keycode that indicates a mnemonic key
`int getHorizontalAlignment()`	It gets the alignment of the label's contents along the `x` axis
`int getHorizontalTextPosition()`	It gets the horizontal position of the label's text, relative to its image
`Icon getIcon()`	It gets the graphic image (glyph or icon) that the label displays
`int getIconTextGap()`	It gets the amount of space between the text and the icon displayed in this label
`Component getLabelFor()`	It gets the component this object is a label for
`String getText()`	It gets the text string that the label displays
`LabelUI getUI()`	It gets the look and feel object that renders this component
`String getUIClassID()`	It gets a string that indicates the name of the look and feel class that renders this component
`int getVerticalAlignment()`	It gets the alignment of the label's contents along the `y` axis

Table 12.2: Methods of the JLabel class

Method	Does this
`int getVerticalTextPosition()`	It gets the vertical position of the label's text, relative to its image
`boolean imageUpdate(Image img, int infoflags, int x, int y, int w, int h)`	It is overridden and returns False if the current Icon's image is not equal to that passed in `Image img`
`protected String paramString()`	It gets a string representation of this `JLabel` object
`void setDisabledIcon (Icon disabledIcon)`	It sets the icon to be displayed if this `JLabel` object is "disabled"
`void setDisplayedMnemonic (char aChar)`	It specifies the displayed mnemonic as a `char` value
`void setDisplayedMnemonic (int key)`	It specifies a keycode that indicates a mnemonic key
`void setDisplayedMnemonicIndex (int index)`	It provides a hint to the look and feel for decorating a character in the text to represent the mnemonic
`void setHorizontalAlignment (int alignment)`	It sets the alignment of the label's contents along the `x`-axis
`void setHorizontalTextPosition (int textPosition)`	It sets the horizontal position of the label's text relative to its image
`void setIcon(Icon icon)`	It defines the icon this component will display
`void setIconTextGap(int iconTextGap)`	It defines the space between them if both the icon and text properties are set
`void setLabelFor (Component c)`	It sets the component this object is labeling
`void setText(String text)`	It defines the single line of text this component will display
`void setUI(LabelUI ui)`	It sets the look and feel object that renders this component
`void setVerticalAlignment (int alignment)`	It sets the alignment of the label's contents along the `y` axis
`void setVerticalTextPosition (int textPosition)`	It sets the vertical position of the text relative to the image
`void updateUI()`	It resets the UI property to a value from the current look and feel

Here's a basic example that shows how to create a Swing label and add a text "Kogent Learning Solutions!" to it:

```
import java.awt.*;
import javax.swing.*;
/*<APPLET
  CODE=JLabelDemo.class
  WIDTH=250
  HEIGHT=200>
</APPLET>*/
public class JLabelDemo extends JApplet {
public JLabelDemo() {
 Container c = getContentPane();
     JLabel jlabel1 = new JLabel("Kogent Learning Solutions!");
      c.add(jlabel1); }
   }
```

The result of this code appears as shown in Figure 12.2:



Figure 12.2: A Swing label

There's a lot more you can do with Swing labels. For example, you can set the alignment of the text in a label as well as use images, and we'll take a look at the possibilities over the next couple of solutions. To display an image in a label, you need an object that implements the `Icon` interface, and you'll find the methods of this interface in Table 12.3:

Table 12.3: Methods of the Icon interface

Method	Does this
`int getIconHeight()`	It gets the icon's height
`int getIconWidth()`	It gets the icon's width
`void paintIcon(Component c, Graphics g, int x, int y)`	It draws the icon at the indicated location

Fortunately, there's an easy way to create icons from images in Swing—you can use the `ImageIcon` class, coming up in the next solution.

Using Image Icons

"Hey," says the NP, "I was just taking a look at the `Icon` interface. Do I have to paint images in icons myself in Swing?" "Of course not," you say. "You can use the `ImageIcon` class to make things a lot easier." "Tell me all about it," the NP says.

The `ImageIcon` class lets you create an icon from an image file for use in a Swing control. Here's the inheritance diagram for this class:

```
java.lang.Object
  |____javax.swing.ImageIcon
```

You'll find the constructors of the `ImageIcon` class in Table 12.4 and its methods in Table 12.5:

Table 12.4: Constructors of the ImageIcon class

Constructor	Does this
`ImageIcon()`	It constructs an unitialized image icon
`ImageIcon(byte[] imageData)`	It constructs an image icon from an array of bytes that were read from an image file containing a supported image format, such as GIF or JPEG
`ImageIcon(byte[] imageData, String description)`	It constructs an image icon from an array of bytes that were read from an image file containing a supported image format, such as GIF or JPEG
`ImageIcon(Image image)`	It constructs an image icon from an `image` object
`ImageIcon(Image image, String description)`	It constructs an image icon from the given image
`ImageIcon(String filename)`	It constructs an image icon from the given file
`ImageIcon(String filename, String description)`	It constructs an image icon from the given file and provide the given description

Table 12.4: Constructors of the ImageIcon class	
Constructor	**Does this**
`ImageIcon(URL location)`	It constructs an image icon from the indicated URL
`ImageIcon(URL location, String description)`	It constructs an image icon from the indicated URL and provide the given description

Table 12.5: Methods of the ImageIcon class	
Method	**Does this**
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this `ImageIcon`
`String getDescription()`	It gets the description of the image
`int getIconHeight()`	It gets the height of the icon
`int getIconWidth()`	It gets the width of the icon
`Image getImage()`	It gets the icon's image
`int getImageLoadStatus()`	It gets the status of the image-loading operation
`ImageObserver getImageObserver()`	It gets the image observer for the image
`protected void loadImage(Image image)`	It loads the image, returning only when the image is loaded
`void paintIcon(Component c, Graphics g, int x, int y)`	It paints the icon
`void setDescription(String description)`	It sets the description of the image
`void setImage(Image image)`	It sets the image displayed by this icon
`void setImageObserver (ImageObserver observer)`	It sets the image observer for the image
`String toString()`	It returns a string representation of this image

The `ImageIcon` class is an extremely useful one in Swing because many Swing components can display icons. We'll put this class to work in the next solution.

Using Images in Labels

The Big Boss appears and says, "We need to jazz things up a bit in our programs; there's some heavy competition in the GUI field now." "OK," you say. "I can add image icons to each label in the code." "Swell," says the BB, "your new code is due yesterday."

You can use several of the `JLabel` constructors to add images to labels, and you can specify the alignment of images and text in a label by using either of the following methods:

- `setVerticalTextPosition()`—It sets the vertical text position relative to the image
- `setHorizontalTextPosition()`—It sets the horizontal text position relative to the image
- `setVerticalAlignment()`—It sets the vertical alignment of the label's contents
- `setHorizontalAlignment()`—It sets the horizontal alignment of the label's contents

You can use constants there to specify the alignments—BOTTOM, CENTER, EAST, HORIZONTAL, LEADING, LEFT, NORTH, NORTH_EAST, NORTH_WEST, RIGHT, SOUTH, SOUTH_EAST, SOUTH_WEST, TOP, TRAILING, VERTICAL, and WEST.

Here's an example in which we add a label with an image and text to an applet. The text will be centered under the image. To create the image, we use the `ImageIcon` class, passing that class's constructor the name of the file that holds the image we want to use (which, in this case, is `Java8.jpg`). We have also used two methods, `setVerticalTextPosition()` and `setHorizontalTextPosition()`, to position the text at the bottom, aligned at the center (as can be seen from figure):

```
    import java.awt.*;
```

```
import javax.swing.*;
/*<APPLET
  CODE=ImageLabelDemo.class
  WIDTH=400
  HEIGHT=200 >
</APPLET>*/
public class ImageLabelDemo extends JApplet {
public void init() {
Container c = getContentPane();
ImageIcon imgIcon = new ImageIcon("Java8.jpg");
JLabel jlabel1 = new JLabel("Kogent Learning Solutions", imgIcon, JLabel.CENTER);
jlabel1.setVerticalTextPosition(JLabel.BOTTOM);
jlabel1.setHorizontalTextPosition(JLabel.CENTER);
c.add(jlabel1); }
}
```

The result appears as shown in Figure 12.3:



Figure 12.3: A Swing label with an image

Note how easy it is to add images to labels in Swing—because it's so simple, it's a very common thing to do.

Using Text Fields

"Don't tell me," the NP says. "I bet I can guess—the name of the Swing text field class is `JTextField`, right?" "Exactly," you smile.

For compatibility, Swing text fields work much like AWT text fields, with some added capabilities. Here's the inheritance diagram for the Swing `JTextField` class:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.text.JTextComponent
                          |____javax.swing.JTextField
```

You'll see more on using Swing text components later in this book, but for now we'll introduce the `JTextField` class so that we can get user input before that point. You'll find the constructors of the `JTextField` class in Table 12.6 and its methods in Table 12.7:

Table 12.6: Constructors of the JTextField class

Constructor	Does this
`JTextField()`	It constructs a new text field
`JTextField(Document doc, String text, int columns)`	It constructs a new `JTextField` object that uses the given storage model and the given number of columns
`JTextField(int columns)`	It constructs a new empty text field with the indicated columns
`JTextField(String text)`	It constructs a new text field initialized with the indicated text
`JTextField(String text, int columns)`	It constructs a new text field initialized with the indicated text and columns

Table 12.7: Methods of the JTextField class

Method	Does this
`protected void actionPropertyChanged (Action action, String propertyName)`	It updates the textfield's state in response to property changes in associated `Action`
`void addActionListener (ActionListener l)`	It adds the action listener to get action events from this text field
`protected void configurePropertiesFromAction(Action a)`	It sets the properties on this textfield to match those in the specified `Action`
`protected PropertyChangeListener createActionPropertyChangeListener (Action a)`	It creates and returns a `PropertyChangeListener` that is responsible for listening changes from a specified `Action` and updating the appropriate properties
`protected Document createDefaultModel()`	It constructs the default implementation of the model
`protected void fireActionPerformed()`	It notifies listeners that have registered for notification on this event type
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this `JTextField`
`Action getAction()`	It gets the currently set `Action` for this `ActionEvent`
`Action[] getActions()`	It gets the command list for the editor
`int getColumns()`	It gets the number of columns
`protected int getColumnWidth()`	It gets the column width
`int getHorizontalAlignment()`	It gets the horizontal alignment of the text
`BoundedRangeModel getHorizontalVisibility()`	It gets the visibility of the text field
`Dimension getPreferredSize()`	It gets the preferred size dimensions needed for this text field
`int getScrollOffset()`	It gets the scroll offset
`String getUIClassID()`	It gets the class ID for a UI
`boolean isValidateRoot()`	It revalidates calls that come from the text field by validating the text field unless the text field is contained within a `JViewport`, in which case it returns False
`protected String paramString()`	It gets a string representation of this text field
`void postActionEvent()`	It processes action events occurring on this text field by sending them to any registered `ActionListener` objects
`void removeActionListener (ActionListener l)`	It removes the indicated action listener so that it no longer receives action events from this text field
`void scrollRectToVisible (Rectangle r)`	It scrolls the field left or right
`void setAction(Action a)`	It sets the `Action` for the `ActionEvent` source.
`void setActionCommand (String command)`	It sets the command string used for action events
`void setColumns (int columns)`	It sets the number of columns and then invalidates the layout
`void setDocument (Document doc)`	It associates the editor with a text document
`void setFont(Font f)`	It sets the current font
`void setHorizontalAlignment (int alignment)`	It sets the horizontal alignment of the text
`void setScrollOffset (int scrollOffset)`	It sets the scroll offset in pixels

Here's a quick example that does nothing more than displaying a text field with the text "Kogent Learning Solutions!" written in red color:

```
import java.awt.*;
import javax.swing.*;
```

```
/* <APPLET
   CODE=JTextFieldDemo.class
   WIDTH=250
   HEIGHT=200>
</APPLET>*/
public class JTextFieldDemo extends JApplet {
JTextField jtextfield1;
public void init() {
        jtextfield1 = new JTextField(15);
Container c = getContentPane();
c.setLayout(new FlowLayout());
c.add(jtextfield1);
jtextfield1.setForeground(Color.RED);
jtextfield1.setText("Kogent Learning Solutions!"); }
}
```

The result of this code appears as shown in Figure 12.4:



Figure 12.4: A Swing TextField

This is only the very beginning of using text components in Swing, but it gets us started and provides an easy way for our programs to display output. For example, we'll connect a text field to a button over the next couple of solutions.

Setting Text Field Alignment

"Uh-Oh," the NP says, "I find it very peculiar to work with Java. I want to set the horizontal alignment of `JTextField`. But it seems this will not work." You smile and say, "No, no it is possible."

The horizontal alignment of `JTextField` can be set in different ways. These can be left justified, leading justified, centered, right justified, or trailing justified. Right/trailing justification is significant when the text field size is more than the size of text inside it.

This can be determined by two methods—`setHorizontalAlignment()` and `getHorizontalAlignment()` methods. The *Leading* is the default justification. Following is an example for setting up a `JTextField` and modifying its horizontal alignment at runtime:

```
import javax.swing.*;
import java.awt.event.*;
public class TextAlignmentDemo {
   public static void main(String[] args) {
          final JTextField jtextfield1 = new JTextField("Press enter", 20);
          jtextfield1.setHorizontalAlignment(JTextField.RIGHT);
          jtextfield1.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
                  int align = jtextfield1.getHorizontalAlignment();
                  if (align == JTextField.LEFT)
                    jtextfield1.setHorizontalAlignment(JTextField.RIGHT);
```

```
                    if (align == JTextField.RIGHT)
                      jtextfield1.setHorizontalAlignment(JTextField.CENTER);
                    if (align == JTextField.CENTER)
                              jtextfield1.setHorizontalAlignment(JTextField.LEFT); }});
            JFrame f = new JFrame("Alignment Demo");
            f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            f.getContentPane().setLayout(new java.awt.FlowLayout());
            f.getContentPane().add(jtextfield1);
            f.setSize(275, 75);
            f.setVisible(true);
            jtextfield1.requestFocus(); }
    }
```

The result of the preceding code is shown in Figure 12.5(a), and the text is displayed as shown in Figure 12.5(b) respectivel.

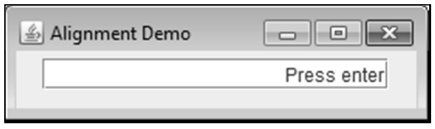


Figure 12.5(a): Showing the TextField

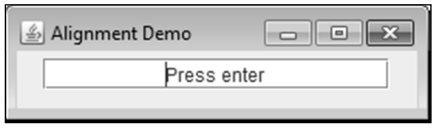


Figure 12.5(b): The text is displayed

Creating Password Field

"So," the NP says, "I can align a text field in various ways…wait a minute…what if I have to set it in such a way that the characters entered are not visible? Is such a thing possible?" You smile again.

The `JPasswordField` class allows you to enter a password in the text field. The characters entered are not shown by default for security purposes. They are masked and a different character, such as an asterisk (*), is displayed in place of each character. The characters are stored in the form of an array instead of a string as a security measure. Just as a normal text field, the password field also fires action event when the user presses Enter inside the password field.

Following is the code which will create and set up the password field:

```
    passwordField = new JPasswordField(10);
    passwordField.setEchoChar('#');
    passwordField.setActionCommand("OK");
    passwordField.addActionListener(this);
```

In this case, the argument which is passed into the `JPasswordField` constructor indicates that the preferred size of the field is 10 (maximum) columns wide. By default, an asterisk "*" character is displayed in the password field. You can also modify the masked character being displayed with any character of your choice, such as "#," by calling the `setEchoChar()` method. This changes the "*" character to "#." Finally, you add the action listener to the password field to validate the password entered. Following is the implementation of the action listener's `actionPerformed()` method:

```
    public void actionPerformed(ActionEvent e) {
            String cmd = e.getActionCommand();
            if (cmd.equals("OK"))  {
      char[] password = passwordField.getPassword();
      if (isPasswordCorrect(password)) {
                    JOptionPane.showMessageDialog(controllingFrame,
            "Your password is correct."); } else {
                    JOptionPane.showMessageDialog(controllingFrame,
                    "Invalid password. Please try again.", "Error Message",
                    JOptionPane.ERROR_MESSAGE); }

            passwordField.selectAll(); resetFocus(); }
    }
```

Although the `getText()` method is available in the `JPasswordField` component, you should avoid using it for security reasons; instead, you should use the `getPassword()` method. Also, in future, it may happen that the `getText()` method may return a visible string (for example, "kogent") ; therefore, it is not very safe. To further enhance the security, as soon as you finish the character array returned by the `getPassword()` method, you should set each of its elements to zero.

Usually, the program validates the password entered before using it to perform any action. In the following code, the validation is performed by calling the `isPasswordCorrect()` method that compares the entered password with the value stored in a different character array:

```
private static boolean isPasswordCorrect(char[] input) {
boolean isCorrect = true;
char[] correctPass = { 'b', 'u', 'g', 'a', 'b', 'o', 'o' };
if (input.length != correctPass.length) { isCorrect = false; }
  else {
        for (int i = 0; i < input.length; i++) {
        if (input[i] != correctPass[i]) { isCorrect = false; }
        }
        }
        for (int i = 0; i < correctPass.length; i++) { correctPass[i] = 0; }
        return isCorrect;
}
```

Creating Text Areas

The NP says, "What is the difference between a `JTextArea` and a `JTextField`? They both look the same...." "There is a distinction," you say, "that too a noticeable one."

A text area is similar to a text field but differs in two respects: it can have multiline input and possess more functionality than a text field. In a `JTextField`, you can append, insert, or replace text at a given location. Since this functionality could be useful for `JTextArea` as well; it is, therefore, a little confusing to try to detect how the distinction is made. You might think that if you want the `JTextArea` functionality everywhere, you can simply use a one-line `JTextArea` in places where you would otherwise use a `JTextField`. In Java 1.0, you also got scroll bars with `JTextArea` even when they were not appropriate. The text areas do not have scroll bars. You have to add the text area to a second component called a scroll pane, created by the `JScrollPane` class. Then you add the scroll pane to the panel, so it can be displayed.

`JTextArea` provides you a component to display multiline text and optionally edit it. A `JTextField` should be preferred to obtain a single line input. However, if the requirement is such that you have to display the text in multiple fonts or styles, then an editor pane or text pane should be preferred. Consider using a label if the displayed text has a limited length and is never edited by the user.

Tables 12.8 and 12.9 list the commonly used `JTextArea` constructors and methods, respectively.

Table 12.8: Constructors of the JTextArea

Constructor	Does this
`JTextArea()`	It constructs a new text area with the empty string as text
`JTextArea(Document doc)`	It constructs a new text area with the given document model, and defaults for all of the other arguments (null, 0, 0).
`JTextArea(Document doc, String text, int rows, int columns)`	It constructs a new text area with the given document model and number of rows and columns
`JTextArea(int rows, int columns)`	It constructs a new text area with the given number of rows and columns and the empty string as text
`JTextArea(String text)`	It constructs a new text area with the specified text
`JTextArea(String text,int rows, int columns)`	It constructs a new text area with the given text, and number of rows and columns

Table 12.9: Methods of the JTextArea

Method	Does this
`void append(String str)`	It appends the given text to the text area's current text
`protected Document createDefaultModel()`	It creates the default implementation of the model to be used at construction
`AccessibleContext getAccessibleContext()`	It returns the `AccessibleContext` associated with this `JTextArea`
`int getColumns()`	It returns the number of columns in this text area
`protected int getColumnWidth()`	It gets the column width
`int getLineCount()`	It gets the number of lines contained in this text area
`int getLineEndOffset (int line)`	It gets the offset of the end of the given line
`boolean getLineWrap()`	It gets the line-wrapping policy of the text area
`Dimension getPreferredScrollableViewportSize()`	It gets the preferred size of the viewport if this component is embedded in a scroll pane
`Dimension getPreferredSize()`	It returns the preferred size of the text area
`protected int getRowHeight()`	It gets the height of the row
`int getRows()`	It gets the number of rows in the text area
`boolean getScrollableTracksViewportWidth()`	It returns True if a viewport should always force the width of this Scrollable to match the width of the viewport
`int getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int direction)`	It states that the components that display logical columns or rows should evaluate the scroll unit increment depending upon the orientation
`int getTabSize()`	It gets the number of characters used to expand tabs
`String getUIClassID()`	It gets the class ID for the UI
`boolean getWrapStyleWord()`	It gets the wrapping style if the text area is wrapping lines
`protected String paramString()`	It returns a string representing the state of this text area
`void replaceRange (String str, int start, int end)`	It replaces text between the indicated start and end positions with the specified replacement text
`void setColumns(int columns)`	It sets the number of columns for this text area
`void setFont(Font f)`	It sets the current font
`void setLineWrap(boolean wrap)`	It sets the line wrapping policy of this text area
`void setRows(int rows)`	It sets the number of rows for this text area
`void setTabSize(int size)`	It sets the number of characters to expand tabs to
`void setWrapStyleWord (boolean word)`	It sets the wrapping style used if the text area is wrapping lines

The code that creates and initializes the text area:

```
area1 = new JTextArea(5,20);
JScrollPane jscrollpane1 = new JScrollPane(area1,
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
area1.setEditable(false);
```

The arguments in the `JTextArea` constructor represent the number of rows and columns in the text area. Also, these arguments play a key role in deciding the size of scroll pane that actually contains the text area. In the absence of the scroll pane, the text area will not allow scrolling the text.

A text area is editable by default, but you can make it uneditable by passing False in the `setEditable(boolean)` method. You can still select and copy the data from a text field, but making any changes to the data directly will not be permitted. The '\n' character is used to write text in the next line; it is called a newline character. The following code depicts how to add text to a text area and note that the text system uses the `'\n'` character internally to represent newlines:

```
    private final static String char = "\n";
    . . .
    area1.append(text + char);
```

By default, a text area automatically scrolls to make visible the appended text. However, automatically scrolling doesn't occur when the user has moved the caret (also called insertion point) to some other point by dragging or clicking in the text area. After you call the `append()` method, moving the caret position to the end of text area scrolls it to the bottom. The `setCaretPosition()` method is used to explicitly set the caret position. This is shown as follows:

```
area1.setCaretPosition(textArea.getDocument().getLength());
```

Customizing a Text Area

A text area can be customized in various ways. For example, you can customize the font and color of text in the text area, determine the number of characters per tab, and define how a text area wraps a line. Being a subclass of `JTextComponent`, `JTextArea` inherits several features from it, such as support for dragging the text, positioning the caret, and many more. The following code depicts an editable text area after initialization:

```
JTextArea area1 = new JTextArea("This is an editable JTextArea. " +
"A text area is a plain text component." +
"Although it can display text in any font, " +
"all of the text inside a text area appears in a one common font.");
area1.setFont(new Font("SansSerif", Font.BOLD, 18));
area1.setLineWrap(true);
area1.setWrapStyleWord(true);
```

A text area doesn't implicitly wrap the lines; instead, it uses only one line to display the whole text. It allows scrolling horizontally if inside a scroll pane. You can call the `setLineWrap()` method to turn on the wrapping of lines and the `setWrapStyleWord()` method to wrap the lines at word boundaries instead of character boundaries. The following code adds a text area to a scroll pane:

```
JScrollPane jscrollPane1 = new JScrollPane(area1);
jScrollPane1.setVerticalScrollBarPolicy(
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
jscrollPane1.setPreferredSize(new Dimension(250, 250));
```

Note that the `JTextArea` constructor in the preceding code doesn't contain any rows and columns as arguments; it limits the text area size by setting the preferred size of scroll pane.

Creating Editor Panes

The NP appears and says, "I need to know how two Swing classes support styled text." "OK," you say, "These come under creating `Editor Panes`. `JEditorPane` and its subclass `JTextPane` will help in this regard."

`JEditorPane` and its subclass `JTextPane` support styled text. `JEditorPane` serves as the foundation for styled text components and provides support for customized text formats. For unstyled text, you may use a text area. The editor pane uses `EditorKit` to mould itself into a sort of text editor and display the content. It's actually the currently installed `EditorKit` that determines the content type displayed by the editor pane. You'll find the constructors of JEditorPane in Table 12.10 and its methods in Table 12.11:

Table 12.10: Constructors of the JEditorPane class

Constructor	Does this
`JEditorPane()`	It creates a new `JEditorPane`
`JEditorPane(String url)`	It creates a `JEditorPane` based on a string containing a URL specification

Table 12.10: Constructors of the JEditorPane class

Constructor	Does this
`JEditorPane(String type, String text)`	It creates a `JEditorPane` that has been initialized to the given text
`JEditorPane (URL initialPage)`	It creates a `JEditorPane` based on a specified `URL` for input

Table 12.11: Methods of the JEditorPane class

Method	Does this
`void addHyperlinkListener (HyperlinkListener listener)`	It adds a hyperlink listener for notification of any changes, for example when a link is selected and entered
`protected EditorKit createDefaultEditorKit()`	It creates the default editor kit (`PlainEditorKit`) when the component is first created
`static EditorKit createEditorKitForContentType(String type)`	It creates a handler for the given type from the default registry of editor kits
`void fireHyperlinkUpdate (HyperlinkEvent e)`	It notifies all listeners that have registered interest for notification on this event type
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this `JEditorPane`
`String getContentType()`	It gets the type of content that this editor is currently set to deal with
`EditorKit getEditorKit()`	It fetches the currently installed kit for handling content
`static String getEditorKitClassNameForContentType (String type)`	It returns the currently registered `EditorKit` class name for the type `type`
`EditorKit getEditorKitForContentType(String type)`	It returns the `EditorKit` to use for the given type of content
`HyperlinkListener[] getHyperlinkListeners()`	It returns an array of all the `HyperlinkListeners` added to this `JEditorPane`
`URL getPage()`	It gets the current `URL` being displayed
`Dimension getPreferredSize()`	It returns the preferred size for the `JEditorPane`
`boolean getScrollableTracksViewportHeight()`	It returns True if a viewport should always force the height of this `Scrollable` to match the height of the viewport
`boolean getScrollableTracksViewportWidth()`	It returns True if a viewport should always force the width of this `Scrollable` to match the width of the viewport
`protected InputStream getStream(URL page)`	It returns a stream for the given `URL`, which is about to be loaded by the `setPage` method
`String getText()`	It returns the text contained in this `TextComponent` in terms of the content type of this editor
`String getUIClassID()`	It gets the class ID for the UI
`protected String paramString()`	It returns a string representation of this `JEditorPane`
`void read(InputStream in, Object desc)`	It initializes from a stream
`static void registerEditorKitForContentType(String type, String classname)`	It establishes the default bindings of `type` to `classname`
`static void registerEditorKitForContentType(String type, String classname, ClassLoader loader)`	It establishes the default bindings of `type` to `classname`
`void removeHyperlinkListener (HyperlinkListener listener)`	It removes a hyperlink listener

Table 12.11: Methods of the JEditorPane class

Method	Does this
`void replaceSelection(String content)`	It replaces the currently selected content with new content represented by the given string
`void scrollToReference (String reference)`	It scrolls the view to the given reference location (that is, the value returned by the `UL.getRef` method for the `URL` being displayed)
`void setContentType (String type)`	It sets the type of content that this editor handles
`void setEditorKit (EditorKit kit)`	It sets the currently installed kit for handling content
`void setEditorKitForContentType(String type , EditorKit k)`	It directly sets the editor kit to use for the given type
`void setPage(String url)`	It sets the current `URL` being displayed
`void setPage(URL page)`	It sets the current `URL` being displayed
`void setText(String t)`	It sets the text of this `TextComponent` to the specified content, which is expected to be in the format of the content type of this editor

If an editor pane displays the content of a new URL, then the content type determines the `EditorKit` you should prefer to load the content. The following different kinds of content are available by default:

- `Text/plain`—It uses an extension of `DefaultEditorKit` to produce a wrapped plain text view.
- `Text/html`—It uses the class `javax.swing.text.html.HTMLEditorKit`, to provide HTML 3.2 support.
- `Text/rtf`—It uses the class `javax.swing.text.rtf.RTFEditorKit`, to provide limited support for the Rich Text Format.

Using HTML in Editor Panes

Displaying text from a URL can be easily done even by an NP. The following code implements this and depicts an uneditable editor pane that displays text formatted with HTML tags:

```
JEditorPane ePane1 = new JEditorPane();
ePane1.setEditable(false);
URL url = DemoTextDisplay.class.getResource("DemoTextDisplay.html");
if (url != null) {
try { ePane1.setPage(url); }
catch (IOException e) {
   System.err.println("Bad URL: " + url); }
   }
   else { System.err.println("Couldn't find file: DemoTextDisplay.html"); }
   JScrollPane jscrollPane1 = new JScrollPane(ePane1);
   jscrollPane1.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
   jscrollPane1.setPreferredSize(new Dimension(250, 145));
   jscrollPane1.setMinimumSize(new Dimension(10, 10));
```

The preceding code uses `JEditorPane's` default constructor to create an editor pane object. After this, the editor pane is made uneditable using `setEditable(false)`. Next, a `URL` class object is obtained and passed as an argument in the `setPage()` method.

The `setPage()` method is important and does the major part of the working. It opens the resource specified by the `URL` and determines the text format (here HTML) used in it. A normal editor pane can display plain text in the form of HTML and Rich-Text-Format (RTF). The GUI may remain responsive if the page loads asynchronously. But that doesn't mean that you count on the data being loaded entirely after the `setPage()` method returns.

Using RTF Files in Editor Panes

The `RTFEditorKit` class is provided for creating RTF text editors. It was developed as a default implementation of `RTF` editing functionality. The Swing team did not write the RTF support. In future, it is expected that the support would improve. The constructor and methods for `RTFEditorKit` are listed in Tables 12.12 and 12.13, respectively.

Table 12.12: Constructor for RTFEditorKit

Constructor	Does this
`RTFEditorKit()`	It constructs an `RTFEditorKit`

Table 12.13: Methods for RTFEditorKit

Method	Does this
`String getContentType()`	It gets the MIME type of the data that this kit represents support for
`void read(InputStream in, Document doc, int pos)`	It inserts content from the given stream which is expected to be in a format appropriate for this kind of content handler
`void read(Reader in, Document doc, int pos)`	It inserts content from the given stream, which will be treated as plain text
`void write (OutputStream out, Document doc, int pos, int len)`	It writes content from a document to the given stream in a format appropriate for this kind of content handler
`void write(Writer out, Document doc, int pos, int len)`	It writes content from a document to the given stream as plain text

Creating Text Panes

The NP says, "Please tell me about `JTextPane` and its significance. Would it support image display?" "Definitely," you say, "let me explain."

The JTextPane is a complete text editor and provides support for almost all the operations of a text editor, such as formatting the text, wrapping word, and displaying images. It takes the assistance of a linked list of objects that implement the Style interface to indicate the formatting style. Additionally, a set of convenient and easy-to-use methods are also provided. For example, you can use the `setPage()` method to load a text pane or an editor pane with text from a particular URL. Unlike `JTextPane`, you can initialize a `JEditorPane` object with a particular URL.

Inserting Images and Controls into Text Panes

You can insert images and controls with the help of two methods. These methods are `JTextPane's insertIcon()` method and `insertComponent()` method. These methods insert the given object by replacing the current selection. And if there is no current selection, the given object will be placed at the beginning of the document. That is why, you defined the `setEndSelection()` method in the example given here to point the selection at the end of the document where we want to do insertion.

When inserting text, you cannot simply append it to the text pane itself. Instead, we retrieve its document and call `insertString()`. To give attributes to inserted text, you can construct `AttributeSet` implementation and assign attributes to that set using the `StyleConstants` class.

In the example given here, you do this by constructing three static instances of `SimpleAttributeSet class`. The `JTextPane` also supports the use of styles, which are named collection of attributes. The code is as follows:

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
```

```
import javax.swing.text.*;
public class JTextPaneDemo extends JFrame {
  static SimpleAttributeSet ITALIC_GRAY = new SimpleAttributeSet();
  static SimpleAttributeSet BOLD_BLACK = new SimpleAttributeSet();
  static SimpleAttributeSet BLACK = new SimpleAttributeSet();
  JTextPane pane1 = new JTextPane();
  static {
        StyleConstants.setForeground(ITALIC_GRAY, Color.gray);
        StyleConstants.setItalic(ITALIC_GRAY, true);
        StyleConstants.setFontFamily(ITALIC_GRAY,  "Helvetica");
        StyleConstants.setFontSize(ITALIC_GRAY, 14);
        StyleConstants.setForeground(BOLD_BLACK, Color.black);
        StyleConstants.setBold(BOLD_BLACK, true);
        StyleConstants.setFontFamily(BOLD_BLACK,  "Helvetica");
        StyleConstants.setFontSize(BOLD_BLACK, 14);
        StyleConstants.setForeground(BLACK, Color.black);
        StyleConstants.setFontFamily(BLACK,"Helvetica");
        StyleConstants.setFontSize(BLACK, 14); }
  public JTextPaneDemo() {
        super ("JTextPane Demo");
        JScrollPane jscrollPane1 = new JScrollPane(pane1);
        getContentPane().add(jscrollPane1, BorderLayout.CENTER);
        setEndSelection();
        pane1.insertIcon(new ImageIcon("logo.jpg"));
        insertText("\nHistory: Distant\n\n", BOLD_BLACK);
        setEndSelection();
        pane1.insertIcon(new ImageIcon("tom.jpg"));
        insertText( "                ", BLACK);
        setEndSelection();
        pane1.insertIcon (new ImageIcon("jerry.jpg"));
        insertText( "\n   Tom &   "+ "Jerry\n\n" , ITALIC_GRAY);
        setEndSelection();
        JButton jbutton1 = new JButton("Visit Home");
        jbutton1.addActionListener (new ActionListener() {
                public void actionPerformed (ActionEvent e) {
                        pane1.setEditable(false);
                        try { pane1.setPage("http://www.logo.com"); }
                                catch(IOException ioe) {
                                        ioe.printStackTrace(); }
                        }});
                pane1.insertComponent(jbutton1);
                setSize (675,500);
                setVisible (true); }
        protected void insertText (String text,AttributeSet set) {
        try {
                pane1.getDocument().insertString(
                pane1.getDocument().getLength(), text, set); }
        catch (BadLocationException e) { e.printStackTrace(); }
  }
  protected void setEndSelection() {
        pane1.setSelectionStart (pane1.getDocument().getLength());
        pane1.setSelectionEnd
        (pane1.getDocument().getLength()); }
  public static void main (String argv[]) { new JTextPaneDemo(); }
}
```

Figure 12.6 shows the result of the code:



Figure 12.6: Showing the images

Setting Text Pane Text Attributes

The NP is little worried about `JTextPane`. He says, "It makes a simple thing difficult." "Yes Sir," you say. "It makes a simple thing difficult, but at the same time, it makes complex things easier."

Swing's `JTextPane` is way better as compared to the AWT's `TextArea`. It not only allows you to create sophisticated combinations of character and paragraph attributes, images, and other components but also to create styled text editors. `JTextPane` makes complex things easy but sometimes makes some simple things complex. For example, setting the default font and color for an AWT text area is simple, but not so easy for a text pane in Swing. The following code depicts an AWT text area that uses 16-point red `SansSerif` font:

```
import java.awt.TextArea;
import java.awt.*;
import java.awt.event.*;
public class ListenerWindow extends WindowAdapter {
public ListenerWindow() {
Frame frame1 = new Frame("TextArea Example");
frame1.setLayout(new BorderLayout());
TextArea area1 = new TextArea();
Font font = new Font("SansSerif", Font.ITALIC, 16);
area1.setFont(font);
area1.setForeground(Color.red);
frame1.add(area1, BorderLayout.CENTER);
frame1.addWindowListener(this);
frame1.setSize(300, 200);
frame1.setVisible(true); }
public void windowClosing(WindowEvent e) { System.exit(0); }
public static void main(String[] args) {
        ListenerWindow listen = new ListenerWindow(); }
}
```

The execution of the program is shown in Figure 12.7:

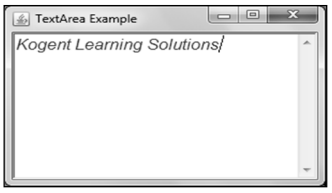


Figure 12.7: Showing AWT TextArea

The preceding code involves call to `TextArea's setFont()` and `setForeground()` methods. The former sets the font style, while the latter sets the text color in the text area. The following code shows a (seemingly) equivalent example using Swing's `JFrame` and `JTextPane` classes:

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class ListenerWindow1 extends WindowAdapter {
public ListenerWindow1() {
JFrame jframe1 = new JFrame("JTextPane Example");
jframe1.getContentPane().setLayout(new BorderLayout());
JTextPane pane1 = new JTextPane();
Font font = new Font("SansSerif", Font.ITALIC, 16);
pane1.setFont(font);
pane1.setForeground(Color.red);
jframe1.getContentPane().add(pane1, BorderLayout.CENTER);
jframe1.addWindowListener(this);
jframe1.setSize(300, 200);
jframe1.setVisible(true); }
public void windowClosing(WindowEvent evt) { System.exit(0); }
public static void main(String[] args) {
  ListenerWindow1 listen = new ListenerWindow1(); }
}
```

The execution of the program is shown in Figure 12.8:

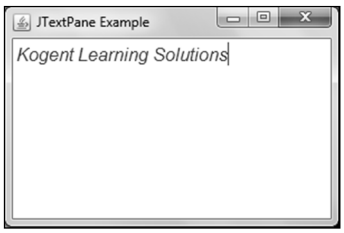


Figure 12.8: Showing the JTextPane

The preceding code doesn't produce the result as expected. The text pane uses the font and color specified by the `setFont()` and `setForeground()` methods.

The idea of setting the font style and color of the whole text seems a bit awkward (contrary to the case of less flexible `JTextArea`) as each character can have its own font and color. Instead, you should look at the whole content and set the character attributes. In case the text pane doesn't contain any text, the attributes set will be used for any text entered later. The following code illustrates the following aspect, where the method `setJTextPaneFont()` is the container for the logic to update the font and color attributes:

```
import javax.swing.*;
import javax.swing.text.*;
import java.awt.event.*;
import java.awt.*;
public class ListenerWindow2 extends WindowAdapter {
public ListenerWindow2() {
JFrame jframe1 = new JFrame("JTextPane Attribute Example");
jframe1.getContentPane().setLayout(new BorderLayout());
JTextPane pane1 = new JTextPane();
Font font = new Font("SansSerif", Font.ITALIC, 16);
setJTextPaneFont(pane1, font, Color.red);
jframe1.getContentPane().add(pane1, BorderLayout.CENTER);
jframe1.addWindowListener(this);
jframe1.setSize(300, 200);
jframe1.setVisible(true); }
public static void setJTextPaneFont(JTextPane pane, Font font, Color c) {
MutableAttributeSet set = pane.getInputAttributes();
StyleConstants.setFontFamily(set, font.getFamily());
StyleConstants.setFontSize(set, font.getSize());
StyleConstants.setItalic(set,(font.getStyle()&Font.ITALIC)!= 0);
```

```
    StyleConstants.setBold(set, (font.getStyle() & Font.BOLD) != 0);
    StyleConstants.setForeground(set, c);
    StyledDocument doc = pane.getStyledDocument();
    doc.setCharacterAttributes(0, doc.getLength() + 1, set, false); }
    public void windowClosing(WindowEvent evt) { System.exit(0); }
    public static void main(String[] args) {
      ListenerWindow2 listener = new ListenerWindow2(); }
    }
```

The execution of the program is shown in Figure 12.9:

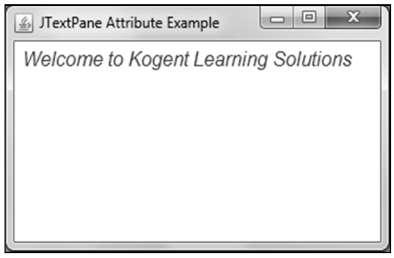


Figure 12.9: The JTextPane attributes

The preceding example produces 16-point red text, "Welcome to Kogent Learning Solutions," within the `JTextPane`.

The `JTextPane` can't be considered a direct replacement for the AWT's `TextArea`. Swing provides the `JTextArea` class to create a behavior similar to the `TextArea` and support source code compatibility. However, you should use a `JTextPane` instead of a `JTextArea` even in cases where multiple font support is not required within the text component. For example, embedded images or alternate text alignment is not supported in `JTextArea`.

Working with Sound in Applets

The NP appears and says, "I want audio to do business with the images." You smile and say, "Certainly... true multimedia presentations often combine image techniques, such as animation with sound. It gives added interest to the user."

By now you have gone through the drawing and image processing, and you are wondering what in the world audio has to do with images. In this regards we are going to introduce a discussion.

`java.applet.AudioClip` defines an interface for objects that can play sound. An object that implements `AudioClip` can be told to `play()` its sound data, `stop()` playing the sound, or `loop()` continually.

An applet can call its `getAudioClip()` method to retrieve sounds over the network. This method takes an absolute or relative URL to specify where the audio file is located. The viewer may take the sound from a cache or retrieve it over the network. The following applet gives a simple example:

```
    import java.awt.*;
    import java.awt.event.*;
    /* <APPLET
      CODE=SoundAppDemo.class
      WIDTH=250
      HEIGHT=200 >
    </APPLET>*/
    public class SoundAppDemo extends java.applet.Applet implements ActionListener
    {
    java.applet.AudioClip sound;
    public void init()
    {
    sound = getAudioClip(getCodeBase(), "sound.au");
    Button play = new Button("Play Sound");
    play.addActionListener(this);
    add(play); }
    public void actionPerformed(ActionEvent e)
```

```
    {
            if(sound != null)
            sound.play(); }
    }
```

The applet retrieves an `AudioClip` from the server. You must note that the applets only support sound files having .au extension.

Working with Sound in Application

The Big Boss is unhappy, "Java comprises very limited options for sound and sound devices." "No Sir," you say. "The initial version of Java has limited options. But now you have Java Media Framework. It enables audio, video, and other time-based media to be added to applications and applets built on Java technology."

Initially, Java had very less support for sound and sound devices. Only `.AU` and `.WAV` files were supported. But as the time passed, Java evolved with stronger multimedia facilities. The Java Sound API, a very low-level API, supported multiple audio operations, such as capture and playback, mixing, MIDI sequencing and synthesis, in a very flexible and extensible framework. It empowered the programmers to experiment with sound the way they like, but it was still hard to program. Several initiatives emerged for digital video and, ultimately, the Java Media Framework (JMF) came to take the sound facilities in Java to a new high.

The JMF's cross-platform, powerful, and scalable toolkit lets you embed audio, video, and other time-based media in various Java applications. You can capture playback, stream, and transcode multiple media formats using the JMF.

Although the JMF added an entirely different capability to Java, yet a lot is to be done. For example, there is still limited support for audio standards, such as MPEG-2 Layer 3 (MP3).

Abstract Button: The Foundation of Swing Buttons

"But I don't want to use the `AbstractButton` class," the Novice Programmer says. "I want to use the `JButton` class. Why do I have to learn about `AbstractButton`?" "All buttons in Swing are derived from `AbstractButton`," you say, "and there's a tremendous number of methods in that class you should know about, even if only for reference. Really, it'll do you good." "Oh," says the NP.

The `AbstractButton` class provides the foundation of the button classes in Java, and like `JComponent`, it provides so many methods that we can't ignore it here. Here's the inheritance diagram for `AbstractButton`:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.AbstractButton
```

You'll find the fields of the `AbstractButton` class in Table 12.14, its constructor in Table 12.15, and its methods in Table 12.16.

Table 12.14: Fields of the AbstractButton class

Field	Does this
`protected ActionListener actionListener`	The button model's `ActionListener`
`static String BORDER_PAINTED_CHANGED_PROPERTY`	It indicates a change to the border painting
`protected ChangeEvent changeEvent`	The button's `ChangeEvent` object (only one `ChangeEvent` is needed per button)
`protected ChangeListener changeListener`	The button's model `ChangeListener`
`static String CONTENT_AREA_FILLED_CHANGED_PROPERTY`	It indicates a change to have the button paint the content area
`static String DISABLED_ICON_CHANGED_ PROPERTY`	It indicates a change to the icon used when the button has been disabled
`static String DISABLED_SELECTED_ICON_CHANGED_PROPERTY`	It indicates a change to the icon used when the button has been disabled and selected

Table 12.14: Fields of the AbstractButton class

Field	Does this
`static String FOCUS_PAINTED_CHANGED_ PROPERTY`	It indicates a change to having the border highlighted when it has the focus (or not)
`static String HORIZONTAL_ALIGNMENT_CHANGED_ PROPERTY`	It indicates a change in the button's horizontal alignment
`static String HORIZONTAL_TEXT_POSITION_CHANGED_ PROPERTY`	It indicates a change in the button's horizontal text position
`static String ICON_CHANGED_PROPERTY`	It indicates a change to the icon that represents the button
`protected ItemListener itemListener`	The `ItemListener` object for this button.
`static String MARGIN_CHANGED_PROPERTY`	It indicates a change to the button's margins
`static String MNEMONIC_CHANGED_PROPERTY`	It indicates a change to the button's mnemonic
`protected ButtonModel model`	The data model that determines the button's state
`static String MODEL_CHANGED_ PROPERTY`	It indicates a change in the button model
`static String PRESSED_ICON_CHANGED_PROPERTY`	It indicates a change to the icon used when the button has been pressed
`static String ROLLOVER_ENABLED_CHANGED_PROPERTY`	It indicates a change in the button's "rollover enabled" property
`static String ROLLOVER_ICON_CHANGED_PROPERTY`	It indicates a change to the icon used when the cursor is over the button
`static String ROLLOVER_SELECTED_ICON_CHANGED_ PROPERTY`	It indicates a change to the icon used when the cursor is over the button and the button is selected
`static String SELECTED_ICON_CHANGED_PROPERTY`	It indicates a change to the icon used when the button has been selected
`static String TEXT_CHANGED_PROPERTY`	It indicates a change in the button's text
`static String VERTICAL_ALIGNMENT_CHANGED_PROPERTY`	It indicates a change in the button's vertical alignment
`static String VERTICAL_TEXT_POSITION_CHANGED_PROPERTY`	It indicates a change in the button's vertical text position

Table 12.15: The constructor of the AbstractButton class

Constructor	Does this
`AbstractButton()`	It constructs a new `AbstractButton` object

Table 12.16: Methods of the AbstractButton class

Method	Does this
`protected void actionPropertyChanged (Action action, String propertyName)`	It updates the button's state as a result of property changes in the associated action
`void addActionListener (ActionListener l)`	It adds an action listener to the button
`void addChangeListener(ChangeListener l)`	It adds a change listener to the button
`protected void addImpl(Component comp, Object constraints, int index)`	It adds the specified component to this container at the specified index
`void addItemListener (ItemListener l)`	It adds an item listener to a checkbox
`protected int checkHorizontalKey(int key, String exception)`	It verifies that `key` is a legal value for the `horizontalAlignment` and `horizontalTextPosition` properties

Table 12.16: Methods of the AbstractButton class

Method	Does this
`protected int checkVerticalKey(int key, String exception)`	It verifies that `key` is a legal value for the vertical properties
`protected void configurePropertiesFromAction(Action a)`	It sets properties on this button to match those in the specified `Action`
`protected ActionListener createActionListener()`	It creates an action listener
`protected PropertyChangeListener createActionPropertyChangeListener (Action a)`	It creates and returns a `PropertyChangeListener` that is responsible for listening for changes from the specified `Action` and updating the appropriate properties
`protected ChangeListener createChangeListener()`	It returns another `ChangeListener` implementation
`protected ItemListener createItemListener()`	It creates an item listener
`void doClick()`	It programmatically clicks the button
`void doClick (int pressTime)`	It programmatically clicks the button at a certain time
`protected void fireActionPerformed (ActionEvent event)`	It notifies all listeners that have registered for notification on this event type
`protected void fireItemStateChanged (ItemEvent event)`	It notifies all listeners that have registered for notification on this event type
`protected void fireStateChanged()`	It notifies all listeners that have registered for notification on this event type
`String getActionCommand()`	It gets the action command for this button
`Action getAction()`	It returns the currently set `Action` for this `ActionEvent`
`Icon getDisabledIcon()`	It gets the disabled icon
`Icon getDisabledSelectedIcon()`	It gets the disabled selected icon
`int getDisplayedMnemonicIndex()`	It returns the character, as an index, that the look and feel should provide decoration for as representing the mnemonic character
`boolean getHideActionText()`	It returns the value of `hideActionText` property, which determines whether the button displays text from the `Action`
`int getHorizontalAlignment()`	It gets the horizontal alignment of the icon and text
`int getHorizontalTextPosition()`	It sets the horizontal position of the text relative to the icon
`Icon getIcon()`	It gets the default icon
`int getIconTextGap()`	It gets the amount of space between the text and the icon displayed in this button
`ItemListener[] getItemListeners()`	It returns an array of all `ItemListeners` added to this `AbstractButton`
`String getLabel()`	It is deprecated and replaced by `getText()`
`Insets getMargin()`	It gets the margin between the button's border and the label
`int getMnemonic()`	It gets the keyboard mnemonic from the current model
`ButtonModel getModel()`	It gets the model that this button represents
`long getMultiClickThreshhold()`	It gets the time in milliseconds required between mouse press events for the button to generate action events
`Icon getPressedIcon()`	It gets the pressed icon for the button
`Icon getRolloverIcon()`	It gets the rollover icon for the button
`Icon getRolloverSelectedIcon()`	It gets the rollover selected icon for the button
`Icon getSelectedIcon()`	It gets the selected icon for the button

Table 12.16: Methods of the AbstractButton class

Method	Does this
`Object[] getSelectedObjects()`	It gets an array (of length 1) containing the label or null if the button is not selected
`String getText()`	It gets the button's text.
`ButtonUI getUI()`	It gets the button's current UI
`int getVerticalAlignment()`	It gets the vertical alignment of the text and icon
`int getVerticalTextPosition()`	It gets the vertical position of the text relative to the icon
`boolean imageUpdate(Image img, int infoflags, int x, int y, int w, int h)`	It returns False if the current icon's image is not equal to that passed in `Image img`
`protected void init (String text, Icon icon)`	For internal use
`boolean isBorderPainted()`	It gets the `borderPainted` property
`boolean isContentAreaFilled()`	It get the `contentAreaFilled` property
`boolean isFocusPainted()`	It gets the `paintFocus` property
`boolean isRolloverEnabled()`	It gets the `rolloverEnabled` property
`boolean isSelected()`	It gets the state of the button
`protected void paintBorder(Graphics g)`	It paints the button's border if `borderPainted` property is True
`protected String paramString()`	It gets a string representation of this `AbstractButton` object
`void removeActionListener (ActionListener l)`	It removes an `ActionListener` from the button
`void removeChangeListener (ChangeListener l)`	It removes a `ChangeListener` from the button
`void removeItemListener (ItemListener l)`	It removes an `ItemListener` from the button
`void removeNotify()`	It notifies this component that it no longer has a parent component
`void setAction(Action a)`	It sets the `Action`
`void setActionCommand (String actionCommand)`	It sets the action command for the button
`void setBorderPainted (boolean b)`	It sets the `borderPainted` property
`void setContentAreaFilled (boolean b)`	It sets the `contentAreaFilled` property
`void setDisabledIcon (Icon disabledIcon)`	It sets the disabled icon for the button
`void setDisabledSelectedIcon (Icon disabledSelectedIcon)`	It sets the disabled selected icon for the button
`void setEnabled(boolean b)`	It enables (or disables) the button
`void setFocusPainted (boolean b)`	It sets the `paintFocus` property
`void setHideActionText(boolean hideActionText)`	It sets the `hideActionText` property
`void setHorizontalAlignment (int alignment)`	It sets the horizontal alignment of the icon and text
`void setHorizontalTextPosition (int textPosition)`	It sets the horizontal position of the text relative to the icon
`void setIcon(Icon defaultIcon)`	It sets the button's default icon
`void setIconTextGap(int iconTextGap)`	It defines the space between icon and text if both the icon as well as text properties are set
`void setLabel (String label)`	It is deprecated and replaced by `setText(text)`
`void setLayout(LayoutManager mgr)`	It sets the layout manager for this component

Table 12.16: Methods of the AbstractButton class	
Method	**Does this**
`void setMargin(Insets m)`	It sets the space for margins
`void setMnemonic (char mnemonic)`	It is obsolete and `setMnemonic(int)` should be preferred
`void setMnemonic (int mnemonic)`	It sets the keyboard mnemonic on the current model
`void setModel(ButtonModel newModel)`	It sets the model that this button represents
`void setMultiClickThreshhold (long threshold)`	It sets the time in milliseconds required between mouse press events for the button to generate action events
`void setPressedIcon (Icon pressedIcon)`	It sets the pressed icon for the button
`void setRolloverEnabled (boolean b)`	It sets the `rolloverEnabled` property
`void setRolloverIcon (Icon rolloverIcon)`	It sets the rollover icon for the button
`void setRolloverSelectedIcon (Icon rolloverSelectedIcon)`	It sets the rollover selected icon for the button
`void setSelected (boolean b)`	It sets the state of the button
`void setSelectedIcon (Icon selectedIcon)`	It sets the selected icon for the button
`void setText(String text)`	It sets the button's text
`void setUI(ButtonUI ui)`	It sets the button's UI
`void setVerticalAlignment (int alignment)`	It sets the vertical alignment of the icon and text
`void setVerticalTextPosition (int textPosition)`	It sets the vertical position of the text relative to the icon
`void updateUI()`	It gets a new UI object from the current look and feel

Using Buttons

The NP arrives and says, "I'm ready to start working with Swing buttons, what can they do?" "Hmm." you say, "Quite a lot! Better pull up a chair."

Swing buttons are supported in the `JButton` class. Here's the inheritance diagram for this class:

```
java.lang.Object
   |____java.awt.Component
         |____java.awt.Container
                |____javax.swing.JComponent
                       |____javax.swing.AbstractButton
                              |____javax.swing.JButton
```

The Swing buttons have more capabilities than their counterparts in AWT, such as you can add tooltip, margins, mnemonics, and images to the Swing buttons. In addition, a Swing button is capable of performing what an AWT button does, such as enabling or disabling, adding an action listener and an action command.

You'll find the constructors of the `JButton` class in Table 12.17 and its methods in Table 12.18. Note that this class is based on the `AbstractButton` class, which means it inherits all of that class's functionality (see the previous solution for the details).

Table 12.17: Constructors of the JButton class	
Constructor	**Does this**
`JButton()`	It creates a button without any icon or text
`JButton(Action a)`	It creates a button where properties are taken from the `Action` supplied
`JButton(Icon icon)`	It creates a button with an icon
`JButton(String text)`	It creates a button with the given text
`JButton(String text, Icon icon)`	It creates a button with the given text and icon

Table 12.18: Methods of the JButton class	
Method	**Does this**
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this `JButton`
`String getUIClassID()`	It gets a string that indicates the name of the look and feel class that renders this component
`boolean isDefaultButton()`	It gets the value of the `defaultButton` property
`boolean isDefaultCapable()`	It gets the value of the `defaultCapable` property
`protected String paramString()`	It gets a string representation of this `JButton`
`void removeNotify()`	It overrides `JComponent.removeNotify` to check whether this button is currently set as the default button on the root pane. If so, it sets the root pane's default button to null to ensure that the root pane doesn't hold onto an invalid button reference
`void setDefaultCapable(boolean defaultCapable)`	It sets the `defaultCapable` property
`void updateUI()`	It resets the UI property to a value from the current look and feel

Let's look at an example that puts the `JButton` class to work. Here, we change the text and font of a label when the user clicks a button. This example is very simple, as you can see in the following code.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
   CODE=JButtonDemo.class
   WIDTH=300
   HEIGHT=200>
</APPLET>*/
public class JButtonDemo extends JApplet {
JButton jbutton1;
JLabel jlabel1;
public void init() {
         jbutton1 = new JButton("Click Me");
         jlabel1 = new JLabel("Hello");
         Container c = getContentPane();
   c.setLayout(new FlowLayout());
   c.add(jlabel1);
   c.add(jbutton1);
   jbutton1.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent event) {
          Font font = new Font("Serif",Font.ITALIC,14);
          jlabel1.setFont(font);
   jlabel1.setText("Kogent"); }});
   }
}
```

The result of this code appears as shown in Figure 12.10:



Figure 12.10: Using a Swing button

As you can see in the figure that when the user clicks the button, the text and font of the text on label changes; initially, it displays "Hello" and later "Kogent."

There's a lot more you can do with the `JButton` objects, and we'll take a look at the possibilities over the following few solutions.

Let's look at another program (`JButtonDemo1.java`) which has some more features—A text field asks for the user's name and then displays a salutation followed by the user's name. For this, it uses `JOptionPane` (a Swing component normally used to display messages) to say good morning to the user. In case the user clicks without entering anything, the program displays an error message again using the `JOptionPane`:

```
import javax.swing.*;
import java.awt.event.*;
public class JButtonDemo1 extends JFrame {
  public static void main(String [] args) { new JButtonDemo1(); }
        private JButton ok;
        private JTextField textName;
        public JButtonDemo1() {
  this.setSize(325,100);
  this.setTitle("Identify yourself");
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  ButtonListener listener = new ButtonListener();
  JPanel jpanel1 = new JPanel();
  jpanel1.add(new JLabel("Enter your name: "));
  textName = new JTextField(15);
  jpanel1.add(textName);
  ok = new JButton("OK");
  ok.addActionListener(listener);
  jpanel1.add(ok);
  this.add(jpanel1);
  this.setVisible(true); }
        private class ButtonListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
  if (e.getSource() == ok) {
  String name = textName.getText();
        if (name.length() == 0) {
  JOptionPane.showMessageDialog(JButtonDemo1.this,
  "Please don't leave the box empty!", "Empty!!",
  JOptionPane.INFORMATION_MESSAGE); }
        else {
  JOptionPane.showMessageDialog(JButtonDemo1.this,
  "Good morning " + textName.getText(),
  "Greetings to you",
  JOptionPane.INFORMATION_MESSAGE); }
        textName.requestFocus(); }
}
}
}
```

The result of this code that appears as shown in Figure 12.11:



Figure 12.11: JOptionPane displaying the message

The figure depicts the state when the user enters a name and clicks the button. In that case, `JOptionPane` displays the message along with the user's name.

Figure 12.12 depicts the state when the user clicks the button leaving the text field empty:



Figure 12.12: JOption displaying message not to leave the box empty

Displaying Images in Buttons

"I know that you can display images in Swing labels," the NP says, "but what about displaying them in buttons?" "No problem at all," you say, "as long as you have an `ImageIcon` object."

It's easy to display images in buttons using the `ImageIcon` class, because several `JButton` constructors let you add icons to buttons. You can also set the alignment of the text and image using the following `AbstractButton` methods:

- `setVerticalTextPosition()`—It sets the vertical text position relative to the image
- `setHorizontalTextPosition()`—It sets the horizontal text position relative to the image
- `setVerticalAlignment()`—It sets the vertical alignment of the button's contents
- `setHorizontalAlignment()`—It sets the horizontal alignment of the button's contents

Let's look at an example in which we add an image from `ImageButtonDemo.jpg` to the button example from the previous solution:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
   CODE= ImageButtonDemo.class
   WIDTH=450
   HEIGHT=200 >
</APPLET>*/
public class ImageButtonDemo extends JApplet {
JButton jbutton1 = new JButton("Button", new ImageIcon("button.jpg"));
JTextField jtext1 = new JTextField(20);
public void init() {
Container c = getContentPane();
c.setLayout(new FlowLayout());
c.add(jbutton1);
c.add(jtext1);
jbutton1.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent event) {
jtext1.setText("Displaying image on a Swing Button"); }});
}
}
```

The result appears as shown in Figure 12.13:



Figure 12.13: Using a Swing button with an image

As you can see in the figure that the button now displays a large image, making the whole program more visually interesting.

The next example displays three buttons with images and a text field. Each button represents an illusion image, which once clicked, displays its name in the text field:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/* <APPLET
  CODE=ImageButtonDemo1.class
  WIDTH=250
  HEIGHT=450>
</APPLET>*/
public class ImageButtonDemo1 extends JApplet
implements ActionListener {
JTextField jtext1;
public void init() {
        try {
        SwingUtilities.invokeAndWait(
        new Runnable() { public void run() { createGUI(); }
        });
        }
  catch (Exception excep) {
  System.out.println("Sorry! Can't be created " + excep); }
        }
        private void createGUI() {
setLayout(new FlowLayout());
ImageIcon Bulge = new ImageIcon("bulg.jpg");
JButton click= new JButton(Bulge);
click.setActionCommand("Optical Illusion - Bulging");
click.addActionListener(this);
add(click);
ImageIcon Wheels = new ImageIcon("wheels.jpg");
click = new JButton(Wheels);
click.setActionCommand("Optical Illusion - Wheels Rotating");
click.addActionListener(this);
add(click);
ImageIcon Zollner = new ImageIcon("illus.jpg");
click = new JButton(Zollner);
click.setActionCommand("Zollner-illusion");
click.addActionListener(this);
```

```
    add(click);
    jtext1 = new JTextField(20);
    add(jtext1); }
            public void actionPerformed(ActionEvent e) {
      jtext1.setText(e.getActionCommand()); }
    }
```

The result appears as shown in Figure 12.14:

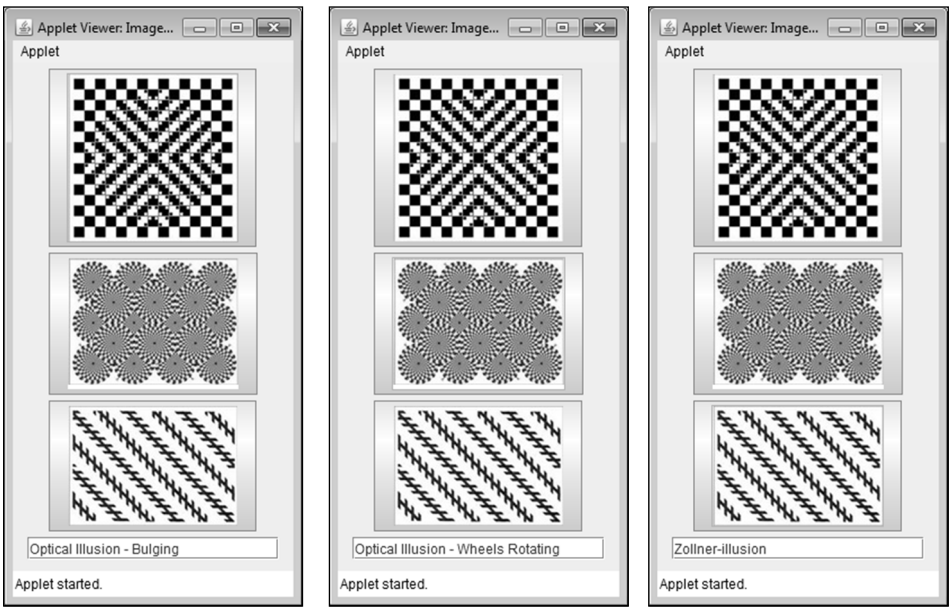


Figure 12.14: Displaying text attached with each button

In the result, whichever button is clicked, a text related to that particular figure is displayed.

Using Rollover and Disabled Images

The Big Boss is distraught and says, "We have got to jazz our programs up!" "OK," you say. "I can add rollover images to the buttons, so they'll display an image when the mouse is over them. Why do we have to jazz things up?" "Because," the Big Boss says, "our competitors have invested years of work and money to make their programs twice as useful as ours." "Shouldn't we do the same?" you ask. The BB says, "Don't be ridiculous."

You can set a number of icons for buttons. Here are the possibilities (rollover icons appear when the mouse moves over the button):

- Normal icon
- Rollover icon
- Rollover selected icon
- Selected icon
- Pressed icon
- Disabled icon
- Disabled selected icon

We'll put these possibilities to work in an example. To add these icons, we just use a method with the matching name: `setIcon()`, `setRolloverIcon()`, `setRolloverSelectedIcon()`, `setSelectedIcon()`, `setPressedIcon()`, `setDisabledIcon()`, or `setDisabledSelectedIcon()`. Here's how this looks in code (note that to enable rollover events, we call `setRolloverEnabled()` with a value of True):

```
import java.awt.*;
import javax.swing.*;
/* <APPLET
   CODE=ButtonIconsDemo.class
   WIDTH=300
   HEIGHT=200>
</APPLET>*/
public class ButtonIconsDemo extends JApplet {
   public void init() {
Container c = getContentPane();
Icon normalIcon = new ImageIcon("rollover 1.jpg");
Icon rolloverIcon = new ImageIcon("rollover 2.jpg");
Icon pressedIcon = new ImageIcon("rollover 3.jpg");
JButton jbutton1 = new JButton();
jbutton1.setRolloverEnabled(true);
jbutton1.setIcon(normalIcon);
jbutton1.setRolloverIcon(rolloverIcon);
jbutton1.setPressedIcon(pressedIcon);
c.setLayout(new FlowLayout());
c.add(jbutton1); }
}
```

You can see the result in Figure 12.15:

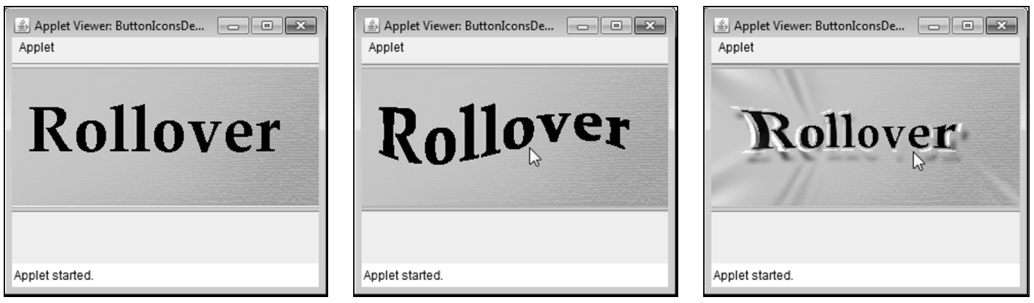


Figure 12.15: Using the rollover image and other images in a button

In the figure, when the mouse moves over the button, the rollover image becomes visible; and when the button is pressed, a pressed image becomes visible.

Default Buttons and Mnemonics

If you have taken a look at dialog boxes for your operating system, you'll see that one button is usually marked as the *default* button, and that button is automatically clicked if the user performs some keyboard action, such as pressing the Enter key. You can make a button the default button in Swing using the `setDefaultButton()` method.

Besides making buttons as default buttons, you can also give each button a *mnemonic*, which is a keyboard shortcut, much like those you see in menus. You underline one (case-insensitive) letter in a button's caption; and when the button has the focus, typing that character activates the button. If the button does not have a caption, typing the meta key (for example, that's the Alt key in Windows) and the button's mnemonic activates the button.

Let's see this in code. In this case, we add two buttons to an applet and make the first button the default button. The first button will have the caption "Button1," and we'll make the letter `B` the button's mnemonic. Here's what this looks like in code (note that we give the root pane the focus at the end of the code so that it can intercept key events and therefore activate the default button when the user presses Enter):

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
```

```
    CODE=DefaultButtonDemo.class
    WIDTH=300
    HEIGHT=200>
</APPLET>*/
public class DefaultButtonDemo extends JApplet {
JButton jbutton1 = new JButton("Button1");
JButton jbutton2 = new JButton("Button2");
JTextField jtext1 = new JTextField(20);
public void init() {
Container c = getContentPane();
c.setLayout(new FlowLayout());
jbutton1.setMnemonic('1');
getRootPane().setDefaultButton(jbutton1);
c.add(jbutton1);
c.add(jbutton2);
c.add(jtext1);
getRootPane().requestFocus();
    jbutton1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
    jtext1.setText("Hello Button1"); }});
    jbutton2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
    jtext1.setText("Hello Button2"); }});
          }
}
```

The result appears as shown in Figure 12.16:

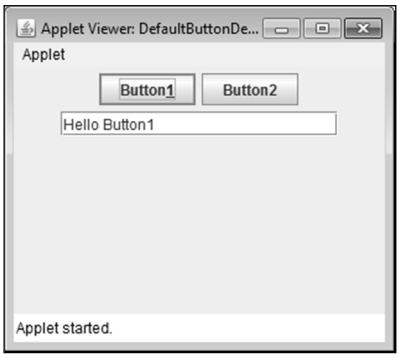


Figure 12.16: Using a default button with a mnemonic

As you can see in the figure, the first button appears with a heavy border, indicating it's the default button, and the '1' in its caption is underlined, indicating that it's the button's mnemonic. When the user presses Enter, the default button is clicked automatically, displaying the message.

Let's take a look at another program that displays two buttons—Press Enter or Click Mouse! and Exit.

```
import javax.swing.*;
import java.awt.event.*;
public class ClickMe extends JFrame {
    public static void main(String [] args) { new ClickMe(); }
          private JButton jbutton1, exitButton;
          public ClickMe() {
          this.setSize(275,100);
          this.setTitle("I am counting!!");
```

```
            this.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
            ClickListener cl = new ClickListener();
            JPanel jpanel1 = new JPanel();
            addWindowListener(new WindowAdapter() {
                    public void windowClosing(WindowEvent e) {
                    exitButton.doClick(); }});
                    jbutton1 = new JButton("Press Enter or Click Mouse!");
                    jbutton1.setMnemonic('P');
                    getRootPane().setDefaultButton(jbutton1);
                    jbutton1.addActionListener(cl);
                    jpanel1.add(jbutton1);
                    exitButton = new JButton("EXIT");
                    exitButton.addActionListener(cl);
                    jpanel1.add(exitButton);
                    this.add(jpanel1);
                    this.setVisible(true); }
            private class ClickListener implements ActionListener {
            private int clickCount = 0;
            public void actionPerformed(ActionEvent e) {
            if (e.getSource() == jbutton1) {
                    clickCount++;
                    if (clickCount == 1)
                    jbutton1.setText("You clicked Me!");
                    else
                    jbutton1.setText("You clicked " + clickCount
                    + " times!");
            }
            else if (e.getSource() == exitButton) {
            if (clickCount > 0)
            System.exit(0);
            else {
                    JOptionPane.showMessageDialog(ClickMe.this,
                    "...try clicking once!!",
                    "In a hurry...!!",
                    JOptionPane.ERROR_MESSAGE); }
            }
            }
      }
   }
```

When the user clicks the "Press Enter or Click Mouse!" button, the button's text changes to "You clicked me!" If the user clicks the button again, the button text changes to "You clicked 2 times!" But if the user presses the Exit button without pressing even once, an error message gets displayed.

The result of the preceding code is shown in Figure 12.17:



Figure 12.17: The ClickMe program counting the clicks

Using Toggle Buttons

The NP is back with a question. "Does Swing add any fundamentally new types of buttons to Java?" "Well," you say, "yes and no." "I knew you'd say that," the NP says. "Toggle buttons are new," you say, "but they are really just the base class of checkboxes and radio buttons." "Tell me more!" the NP says.

Toggle buttons are new in Swing, and they are presented as two-state buttons (actually three states if you count the disabled state) that can appear as selected or deselected. Here's the inheritance diagram for `JToggleButton`, the toggle button class:

```
java.lang.Object
   |____java.awt.Component
         |____java.awt.Container
                |____javax.swing.JComponent
                       |____javax.swing.AbstractButton
                              |____javax.swing.JToggleButton
```

You'll find the constructors of the `JToggleButton` class in Table 12.19 and its methods in Table 12.20:

Table 12.19: Constructors of the JToggleButton class

Constructor	Does this
`JToggleButton()`	It constructs an unselected toggle button without any icon or text
`JToggleButton(Action a)`	It constructs a toggle button with properties taken from the `Action` supplied
`JToggleButton(Icon icon)`	It constructs an unselected toggle button with the indicated image but no text
`JToggleButton(Icon icon, boolean selected)`	It constructs a toggle button with the indicated image and selection state, but no text
`JToggleButton(String text)`	It constructs an unselected toggle button with the indicated text
`JToggleButton(String text, boolean selected)`	It constructs a toggle button with the indicated text and selection state
`JToggleButton(String text, Icon icon)`	It constructs a toggle button that has the indicated text and image and that's initially unselected
`JToggleButton(String text, Icon icon, boolean selected)`	It constructs a toggle button with the indicated text, image, and selection state

Table 12.20: Methods of the JToggleButton class

Method	Does this
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this `JToggleButton`
`String getUIClassID()`	It gets a string holding the name of the look and feel class that renders this component
`protected String paramString()`	It gets a string representation of this toggle button
`void updateUI()`	It resets the UI property to a value from the current look and feel

We'll put `JToggleButton` to use in an example. In this example, we draw a few toggle buttons, most with icons and some with text. Here's what the code looks like:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

/* <APPLET
   CODE=JToggleButtonDemo.class
   WIDTH=420
   HEIGHT=650>
```

```
</APPLET>*/

public class JToggleButtonDemo extends JApplet
{
   public JToggleButtonDemo()
   {
          Container c = getContentPane();
          Icon icon = new ImageIcon("toggle.jpg");
          JToggleButton jtbutton1 = new JToggleButton(icon);
          JToggleButton jtbutton2 = new JToggleButton(icon, true);
          JToggleButton jtbutton3 = new JToggleButton("Click me to toggle!");
          JToggleButton jtbutton4 = new JToggleButton("Click me to toggle!!", icon);
          JToggleButton jtbutton5 = new JToggleButton("Click me to toggle!", icon,true);

          c.setLayout(new FlowLayout());
          c.add(jtbutton1);
          c.add(jtbutton2);
          c.add(jtbutton3);
          c.add(jtbutton4);
          c.add(jtbutton5);
   }
}
```

You can see the result of this code in Figure 12.18:

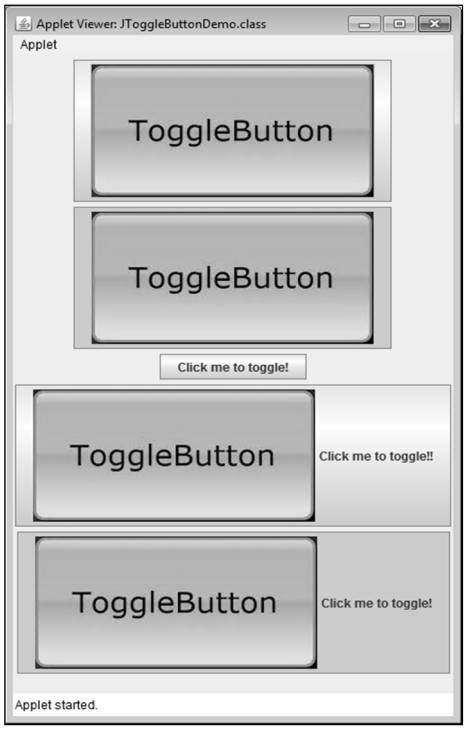


Figure 12.18: Using toggle buttons

As you can see in the figure, you can set the state of a toggle button by passing a value of *true* to its constructor if you want it to appear selected initially.

Creating Toggle Button Groups

The Novice Programmer says, "Toggle buttons are new in Swing—do they do anything new besides presenting the user with a selected or deselected state?" "Sure," you say. "You can also put them together into groups."

You can group toggle buttons as you do other kinds of buttons—with the `ButtonGroup` class. This means that only one of the buttons in a group can be selected at once, just as with radio buttons. Here's an example in which we add an array of five toggle buttons to a group:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
   CODE=ToggleGroupDemo.class
   WIDTH=300
   HEIGHT=600 >
</APPLET>*/
public class ToggleGroupDemo extends JApplet {
public ToggleGroupDemo() {
Container c = getContentPane();
ButtonGroup bg = new ButtonGroup();
JToggleButton[] jtbuttons = new JToggleButton[] {
new JToggleButton(new ImageIcon("toggle.jpg")),
new JToggleButton(new ImageIcon("toggle.jpg")),
new JToggleButton(new ImageIcon("toggle.jpg")),
new JToggleButton(new ImageIcon("toggle.jpg")),
new JToggleButton(new ImageIcon("toggle.jpg")) };
c.setLayout(new FlowLayout());
for(int loop_index=0; loop_index < jtbuttons.length; ++loop_index) {
bg.add(jtbuttons[loop_index]);
c.add(jtbuttons[loop_index]); }
}
}
```

The result appears as shown in Figure 12.19:



Figure 12.19: Using toggle buttons in a group

The toggle buttons in the figure are indeed acting as a group; therefore, when you click one, any other toggle button that was selected is deselected automatically.

Using Checkboxes

The NP says, "I need some way for the user to select an option, from many options. In fact, I need some way for the user to select multiple options from many options. I need...." "Checkboxes," you say. "What you need is checkboxes." "Right," says the NP.

The `JCheckBox` class has some advantages over the AWT `CheckBox` class, such as being able to display images. Here's the inheritance diagram for the `JCheckBox` class:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.AbstractButton
                          |____javax.swing.JToggleButton
                                |____javax.swing.JCheckBox
```

You'll find the constructors for the `JCheckBox` class in Table 12.21 and the methods of this class in Table 12.22:

Table 12.21: Constructors of the JCheckBox class

Constructor	Does this
`JCheckBox()`	It constructs an unselected checkbox without any icon or text
`JCheckBox(Action a)`	It constructs a checkbox with properties taken from the `Action` supplied
`JCheckBox(Icon icon)`	It constructs an unselected checkbox with an icon
`JCheckBox(Icon icon, boolean selected)`	It constructs a checkbox with an icon and indicates whether it's initially selected
`JCheckBox(String text)`	It constructs an unselected checkbox with the given text
`JCheckBox(String text, boolean selected)`	It constructs a checkbox with the given text and indicates whether it's initially selected
`JCheckBox(String text, Icon icon)`	It constructs an unselected checkbox with the given text and icon
`JCheckBox(String text, Icon icon, boolean selected)`	It constructs a checkbox with the given text and an icon and indicates whether it's initially selected

Table 12.22: Methods of the JCheckBox class

Method	Does this
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` associated with this `JCheckBox`
`String getUIClassID()`	It gets a string that indicates the name of the look and feel class that renders this component
`boolean isBorderPaintedFlat()`	It gets the value of the `borderPaintedFlat` property
`protected String paramString()`	It gets a string representation of this checkbox
`void setBorderPaintedFlat (boolean b)`	It sets the `borderPaintedFlat` property, which is used to give a hint to the look and feel as to the appearance of the checkbox border
`void updateUI()`	It resets the UI property to a value from the current look and feel

TIP

In future Java releases, checkboxes will support HTML text.

Let's take a look at an example. Here, we just display four checkboxes and indicate which one the user clicked (note that, as with AWT checkboxes, `ItemListener` is used with `JCheckBox` objects, not `ActionListener`):

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
```

```
      CODE=JCheckBoxDemo.class
      WIDTH=300
      HEIGHT=200 >
    </APPLET>*/
    public class JCheckBoxDemo extends JApplet implements ItemListener {
      JCheckBox jcheck1, jcheck2, jcheck3, jcheck4;
            public void init() {
                    Container c = getContentPane();
                    c.setLayout(new FlowLayout());
                    jcheck1 = new JCheckBox("Red");
                    jcheck2 = new JCheckBox("Green");
                    jcheck3 = new JCheckBox("Blue");
                    jcheck4 = new JCheckBox("Gray");
                    jcheck1.addItemListener(this);
                    jcheck2.addItemListener(this);
                    jcheck3.addItemListener(this);
                    jcheck4.addItemListener(this);
                    c.add(jcheck1);
                    c.add(jcheck2);
                    c.add(jcheck3);
                    c.add(jcheck4);
    }
      public void itemStateChanged(ItemEvent e) {
            if (e.getItemSelectable() == jcheck1) {
                    showStatus("You clicked Red color checkbox");  }
            else if (e.getItemSelectable() == jcheck2) {
                    showStatus("You clicked Green color checkbox");  }
            else if (e.getItemSelectable() == jcheck3) {
                    showStatus("You clicked Blue color checkbox");  }
            else if (e.getItemSelectable() == jcheck4) {
                    showStatus("You clicked Gray color checkbox");  }
      }
    }
```

You can see the result as shown in Figure 12.20:



Figure 12.20: Using checkboxes

As you can see in the figure, the applet reports on its status bar which checkbox has been clicked when the user clicks one. There's more to using checkboxes in Swing, which you'll learn in the solutions following the next one on radio buttons.

Let's take another example of checkboxes, where a user is given the option of choosing pizza toppings of his choice.

```
    import javax.swing.*;
    import java.awt.event.*;
    import javax.swing.border.*;
    public class JCheckBoxDemo1 extends JFrame {
    public static void main(String [] args) { new JCheckBoxDemo1(); }
      private JButton ok;
      private JCheckBox pepperoni, mushrooms, anchovies, Extra, onion, sausage;
      public JCheckBoxDemo1() { this.setSize(600,140);
      this.setTitle("Pizza toppings");
```

```
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        ButtonListener listener = new ButtonListener();
        JPanel mainPanel = new JPanel();
        JPanel sizePanel = new JPanel();
        mainPanel.add(sizePanel);
        JPanel topPanel = new JPanel();
        Border b2 = BorderFactory.createTitledBorder("Varieties available");
        topPanel.setBorder(b2);
        anchovies = new JCheckBox("Anchovies");
        topPanel.add(anchovies);
        Extra = new JCheckBox("Extra cheese");
        topPanel.add(Extra);
        pepperoni = new JCheckBox("Pepperoni");
        topPanel.add(pepperoni);
        onion = new JCheckBox("Onion");
        topPanel.add(onion);
        mushrooms = new JCheckBox("Mushrooms");
        topPanel.add(mushrooms);
        sausage = new JCheckBox("Sausage");
        topPanel.add(sausage);
        mainPanel.add(topPanel);
        ok = new JButton("OK");
        ok.addActionListener(listener);
        mainPanel.add(ok);
        this.add(mainPanel);
        this.setVisible(true); }
        private class ButtonListener implements ActionListener {
              public void actionPerformed(ActionEvent e) {
                      if (e.getSource() == ok) {
                              String tops = "";
                      if (pepperoni.isSelected())
                              tops += "Pepperoni\n";
                      if (mushrooms.isSelected())
                              tops += "Mushrooms\n";
                      if (anchovies.isSelected())
                              tops += "Anchovies\n";
                      if (onion.isSelected()) { tops += "Onion\n"; }
                      if (sausage.isSelected()) { tops += "Sausage\n"; }
                      if (Extra.isSelected())
                              tops += "Extra cheese\n";
                              String msg = "You ordered ";
                      if (tops.equals("")) { msg += "no toppings."; }
                              else { msg += ":\n" + tops; }
                              JOptionPane.showMessageDialog(ok,
                              msg, "Your Order",JOptionPane.INFORMATION_MESSAGE);
                      pepperoni.setSelected(false);
                      mushrooms.setSelected(false);
                      anchovies.setSelected(false);
                      sausage.setSelected(false);
                      Extra.setSelected(false);
                      onion.setSelected(false); }
              }
        }
    }
```

When the user does not select any of the toppings from the list, a message box gets displayed. In case he/she selects some, the same message box details the chosen items. You can see the result of the code in Figure 12.21:

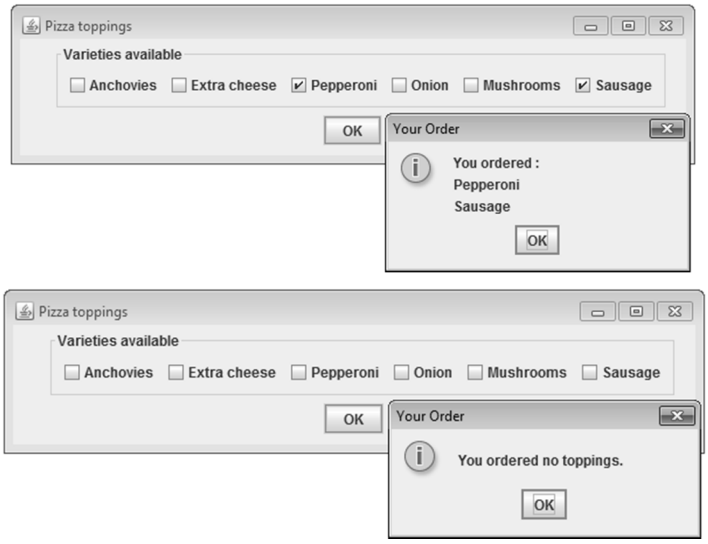


Figure 12.21: The toppings example in progress

TIP

The example JCheckBoxDemo1.java uses the isSelected() method. You can read more about this method under the topic "Getting and Setting the State of Checkboxes and Radio Buttons."

Using Radio Buttons

"OK," says the NP. "I need some radio buttons in my Swing code. Do I make them using the `JCheckBox` class?" "Nope," you say. "Although you use the `CheckBox` class to make radio buttons in AWT programming, radio buttons have their own class in Swing—`JRadioButton`." "Cool!" says the NP.

Here's the inheritance diagram for the `JRadioButton` class:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.AbstractButton
                          |____javax.swing.JToggleButton
                                |____javax.swing.JRadioButton
```

Unlike in AWT programming, radio buttons have their own class in Swing—the `JRadioButton` class. You can find the constructors of the `JRadioButton` class in Table 12.23 and its methods in Table 12.24:

TIP

In future Java releases, radio buttons will support HTML text.

Table 12.23: Constructors of the JRadioButton class

Constructor	Does this
`JRadioButton()`	It constructs an unselected radio button with no set text
`JRadioButton(Action a)`	It constructs a radio button with properties taken from the `Action` supplied
`JRadioButton(Icon icon)`	It constructs an unselected radio button with the given image but no text

Table 12.23: Constructors of the JRadioButton class	
Constructor	**Does this**
`JRadioButton(Icon icon, boolean selected)`	It constructs a radio button with the given image and selection state, but no text
`JRadioButton(String text)`	It constructs an unselected radio button with the given text
`JRadioButton(String text, boolean selected)`	It constructs a radio button with the given text and selection state
`JRadioButton(String text, Icon icon)`	It constructs an unselected radio button that has the given text and image
`JRadioButton(String text, Icon icon, boolean selected)`	It constructs a radio button that has the given text, image, and selection state

Table 12.24: Methods of the JRadioButton class	
Method	**Does this**
`AccessibleContext getAccessibleContext()`	It gets the AccessibleContext associated with this `JRadioButton`
`String getUIClassID()`	It gets the name of the look and feel class that renders this component
`protected String paramString()`	It gets a string representation of this radio button
`void updateUI()`	It resets the UI property to a value from the current look and feel

Let's take a look at an example. Here, we just display four radio buttons and put them into the same button group so that only one of the four can be selected at a time. Here's what the code looks like:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
CODE= JRadioButtonDemo.class
WIDTH=300
HEIGHT=200 >
</APPLET> */
public class JRadioButtonDemo extends JApplet implements ItemListener {
JRadioButton jradio1, jradio2, jradio3, jradio4;
ButtonGroup bg;
public void init() {
Container c = getContentPane();
c.setLayout(new FlowLayout());
bg = new ButtonGroup();
jradio1 = new JRadioButton("Red");
jradio2 = new JRadioButton("Green");
jradio3 = new JRadioButton("Blue");
jradio4 = new JRadioButton("Gray");
bg.add(jradio1);
bg.add(jradio2);
bg.add(jradio3);
bg.add(jradio4);
jradio1.addItemListener(this);
jradio2.addItemListener(this);
jradio3.addItemListener(this);
jradio4.addItemListener(this);
c.add(jradio1);
c.add(jradio2);
c.add(jradio3);
c.add(jradio4);
}
public void itemStateChanged(ItemEvent e) {
```

```
    if (e.getItemSelectable() == jradio1) {
       showStatus("You selected Red color"); }
    else if (e.getItemSelectable() == jradio2) {
       showStatus("You selected Green color"); }
    else if (e.getItemSelectable() == jradio3) {
       showStatus("You selected Blue color"); }
    else if (e.getItemSelectable() == jradio4) {
       showStatus("You selected Gray color"); }
       }
    }
```

You can see the result of this code in Figure 12.22:

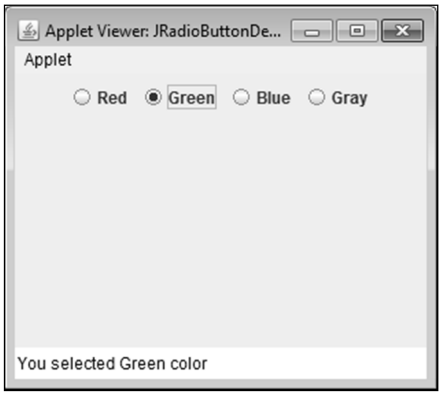


Figure 12.22: Using radio buttons

All the radio buttons in the figure act together as a part of one group. When you click one, any other radio button that was selected in the group becomes deselected.

Let's take the previous example of pizza and update it by adding radio buttons to let the user decide on the size of the pizza as well:

```
    import javax.swing.*;
    import java.awt.event.*;
    import javax.swing.border.*;
    public class JRadioButtonDemo1 extends JFrame  {
             public static void main(String [] args) { new JRadioButtonDemo1(); }
      private JButton ok;
      private JRadioButton small, medium, large;
      private JCheckBox pepperoni, mushrooms, anchovies, Extra, onion, sausage;
      public JRadioButtonDemo1() {
    this.setSize(600,200);
    this.setTitle("Pizza toppings");
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    ButtonListener listener = new ButtonListener();
    JPanel mainPanel = new JPanel();
    JPanel sizePanel = new JPanel();
    Border b1 = BorderFactory.createTitledBorder("Sizes available");
    sizePanel.setBorder(b1);
    ButtonGroup sizeGroup = new ButtonGroup();
    small = new JRadioButton("Small");
    small.setSelected(true);
    sizePanel.add(small);
    sizeGroup.add(small);
    medium = new JRadioButton("Medium");
    sizePanel.add(medium);
```

```
    sizeGroup.add(medium);
    large = new JRadioButton("Large");
    sizePanel.add(large);
    sizeGroup.add(large);
    mainPanel.add(sizePanel);
    JPanel topPanel = new JPanel();
    Border b2 = BorderFactory.createTitledBorder("Varieties available");
    topPanel.setBorder(b2);
    anchovies = new JCheckBox("Anchovies");
    topPanel.add(anchovies);
    Extra = new JCheckBox("Extra cheese");
    topPanel.add(Extra);
    pepperoni = new JCheckBox("Pepperoni");
    topPanel.add(pepperoni);
    onion = new JCheckBox("Onion");
    topPanel.add(onion);
    mushrooms = new JCheckBox("Mushrooms");
    topPanel.add(mushrooms);
    sausage = new JCheckBox("Sausage");
    topPanel.add(sausage);
    mainPanel.add(topPanel);
    ok = new JButton("OK");
    ok.addActionListener(listener);
    mainPanel.add(ok);
    this.add(mainPanel);
    this.setVisible(true); }
            private class ButtonListener implements ActionListener {
      public void actionPerformed(ActionEvent e) {
            String msg = "", tops = "";
     if (e.getSource() == ok) { tops = ""; }
                    if (pepperoni.isSelected()) { tops += "Pepperoni\n"; }
                    if (mushrooms.isSelected()) { tops += "Mushrooms\n"; }
                    if (anchovies.isSelected()) { tops += "Anchovies\n"; }
                    if (onion.isSelected()) { tops += "Onion\n"; }
                    if (sausage.isSelected()) { tops += "Sausage\n"; }
                    if (Extra.isSelected()) {
                                tops += "Extra cheese\n";
      msg = "You ordered "; }
                    if (small.isSelected()) { msg += "A small pizza with "; }
                    if (medium.isSelected()) { msg += "A Medium pizza with "; }
                    if (large.isSelected()) { msg += "A Large pizza with "; }
                    if (tops.equals("")) { msg += "no toppings."; }
                    else { msg += "the toppings:\n" + tops; }
                    JOptionPane.showMessageDialog(ok, msg, "Your Order",
                    JOptionPane.INFORMATION_MESSAGE);
                    pepperoni.setSelected(false);
                    mushrooms.setSelected(false);
                    anchovies.setSelected(false);
                    small.setSelected(true); }
      }
    }
```

The result of the code is displayed in Figure 12.23:

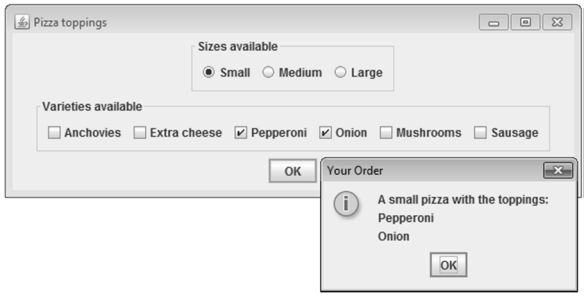


Figure 12.23: Using Radio button and Checkboxes together

TIP

The example JRadioButtonDemo1.java uses the isSelected() method. You can read more about this method under the topic "Getting and Setting the State of Checkboxes and Radio Buttons."

Using Checkbox and Radio Button Images

"Uh-oh," the NP says, "Java's gone all wacky again. We added an image to a checkbox, and now it doesn't work anymore." You smile and say, "That's because you also have to add a selected image to the checkbox. Otherwise, the checkbox's appearance won't change when you click it."

Now that you can add images to checkboxes and radio buttons, there's one important point to know: If you use images in checkboxes or radio buttons, there's no visual indication in the control (such as a checkmark) to show whether it's selected, so you must add a selected image to the control.

Here's an example showing how this works. Here, we add a selected image to a checkbox:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
   CODE=CheckBoxImageDemo.class
   WIDTH=500
   HEIGHT=200>
</APPLET>*/
public class CheckBoxImageDemo extends JApplet implements ItemListener {
JCheckBox jcheck1;
public void init() {
Container c = getContentPane();
c.setLayout(new FlowLayout());
jcheck1 = new JCheckBox("CheckBox 1", new ImageIcon("rollover 4.jpg"));
jcheck1.setSelectedIcon(new ImageIcon("rollover 5.jpg"));
jcheck1.addItemListener(this);
c.add(jcheck1);
}
public void itemStateChanged(ItemEvent e) {
if (e.getItemSelectable() == jcheck1) {
   showStatus("You clicked checkbox 1"); }
}
}
```

You can see the result in Figure 12.24:

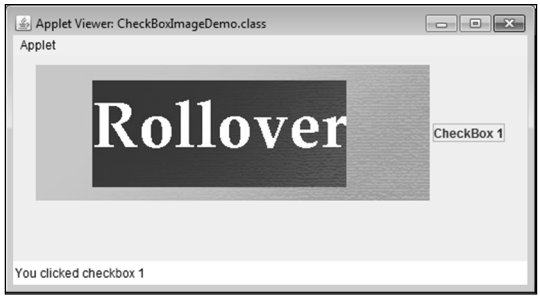


Figure 12.24: Using images in checkboxes

When the user clicks the checkbox, the selected image appears as shown in the figure.

Getting and Setting the State of Checkboxes and Radio Buttons

It's easy enough to respond to checkbox and radio button events, but there are times when you want to work with those controls outside event-handling methods. For example, the user may select a number of options using checkboxes in a dialog box that don't go into effect until he/she dismisses the dialog box, at which time, you need to determine which checkboxes are selected.

You can use the `isSelected()` method to determine whether a checkbox or radio button is selected and the `setState()` method to set the state of a checkbox or radio button. Here's an example in which we display four checkboxes and list which ones are checked when the user clicks any of them. We start by creating an array of four checkboxes and displaying them:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
   CODE=StateDemo.class
   WIDTH=400
   HEIGHT=90 >
</APPLET> */
public class StateDemo extends JApplet implements ItemListener {
JCheckBox checkbox[];
public void init() {
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        checkbox = new JCheckBox[5];
        for(int loop_index = 1; loop_index <= checkbox.length - 1; loop_index++) {
        checkbox[loop_index] = new JCheckBox("CheckBox " + loop_index);
        checkbox[loop_index].addItemListener(this);
        c.add(checkbox[loop_index]); }
        }
        . . .
        . . .
}
```

Now when the user clicks a checkbox, we loop over all the checkboxes, using the `isSelected()` method to see whether they are selected, and list the checkboxes that are selected:

```
public void itemStateChanged(ItemEvent ie) {
  String msg = new String("You selected:\n");
  for(int loop_index = 1; loop_index <= checkbox.length - 1; loop_index++) {
```

```
        if(checkbox[loop_index].isSelected()) {
        msg += " CheckBox " + loop_index; }
        }
        showStatus(msg);
    }
```

The result appears as shown in Figure 12.25:

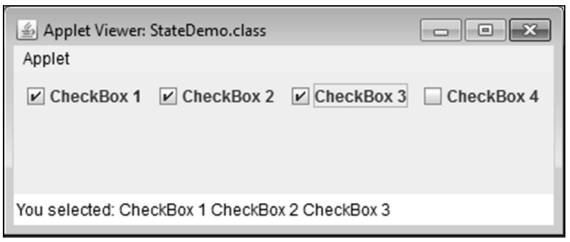


Figure 12.25: Determining which checkboxes are selected

As you can see in the figure, when the user selects checkboxes, the applet indicates which ones are checked.

Summary

In this chapter, we provided you with a brief introduction of labels, password field, text area, editor pane, text pane, buttons, toggle buttons, checkboxes and radio buttons. You learned about the use of labels, image icons, and how to set text field alignment. Then, you learned how to create password field, editor panes, HTML in editor panes, and use of RTF files in editor panes. Later, we explored how to create and insert images in text panes and set text pane attributes. We explored the work with sound and use of buttons.

In the next chapter, you will learn about Swing-Viewports, Scrolling, Sliders, Lists, Tables, and Trees.



13

Swing—Viewports, Scrolling, Sliders, Lists, Tables, and Trees

If you need an immediate solution to:	*See page:*
Handling Viewports	447
Creating Scroll Panes	450
Creating Scroll Pane Headers and Borders	454
Scrolling Images	455
Creating Sliders	456
Filling a Slider	459
Painting Slider Tick Marks	460
Painting Slider Labels	460
Setting the Slider Extent	461
Creating Scrollbars	462
Creating Lists	465
Handling Multiple List Selections	469
Displaying Images in Lists	470
Creating a Custom List Model	471
Creating a Custom List Cell Renderer	472
Handling Double Clicks in Lists	472
Drag and Drop Support	473
Creating a Table	476

If you need an immediate solution to:	*See page:*
Table Sorting and Filtering	478
Adding Rows and Columns to Tables at Runtime	482
Creating Trees	483
Adding Data to Trees	484
Handling Tree Events	486

In Depth

This chapter covers some important Swing topics, namely, Viewports, Scroll panes, Sliders, Scrollbars, List controls, Tables, and Trees. The controls that are discussed in this chapter have one thing in common—scrolling. The list control, in particular, is a very important control in Swing, but it does not handle scrolling by itself; therefore, you need to understand how to work with scrolling before working with lists and many other Swing controls. Let us take a quick look at the topics in this chapter before digging into the code.

Viewports

The `JViewport` class is at the heart of scrolling in Swing. A *viewport* is a window into a view, displaying a section of your data. You can scroll viewports by yourself. Using viewports, you can move around in your displayed data, much like handling the scrolling by yourself. We'll take a look at using the `JViewport` class to scroll images.

Scroll Panes

A common way to implement scrolling in Swing is to use scroll panes because you can scroll components by using scroll panes. A number of Swing controls, such as the `JList` control, implement the `Scrollable` interface to work with scroll panes. In fact, scroll panes are commonly used with `JList` controls to create scrolling lists.

Sliders

Another scrollable control is the Swing slider control, which is supported by the `JSlider` class. Sliders are much like the controls you see in audio devices that let you slide a knob along a track. In fact, sliders are similar to scrollbars, except that you explicitly use sliders to let the user select a value from a continuous range. You can, of course, do the same with scrollbars, but sliders were introduced because users expected that scrollbars will be used to scroll other controls, such as text areas.

Scrollbars

Every Graphical User Interface (GUI) user generally knows about scrollbars, and Swing supports them just as the Abstract Window Toolkit (AWT) does. We will use the `JScrollBar` class in this chapter, scrolling text around an applet. When the scroll box (also called the *thumb* or *bubble*) is moved, the value of the scrollbar changes. You can click the arrow buttons at the end of the scrollbar to change the value of the scrollbar by block increment, or you can click the track of the scrollbar to change its value by unit increment.

Lists

`Lists`, supported by the `JList` class in Swing, are very popular controls because they let you present a list of items in an easy way and hide a long list of items by making the list control scrollable. We'll show you how to scroll long lists as well as cover a number of other Swing topics. For example, you can make multiple selections in various ways in Swing list boxes. You can also display images, handle double and even triple clicks, and more. We'll take a look at how to implement a new model for lists and cell renderers in order to handle the actual display of each item in a list in a customized way.

Tables

The tables in Swing are important and powerful. In principle, they were created to constitute an interface related to database and avoid the inherent complexity of handling data, thus providing flexibility to the programmer. Nevertheless, it's also possible to create a relatively simple `JTable` if it understands the operation correctly. The `JTable` component is flexible enough to show and optionally allow you to edit the tabular data. It presents a simple view of the data without any caching.

Trees

Swing provides the `JTree` class to display data organized in the form of a tree. Each element in the tree is called the `JTree` class and defines a component that displays the data arranged in a tree-like structure. Each element in a tree is called a *node*, and the base node of a tree is referred to as the *root node*. A parent node is a node that has other nodes attached to it, which are known as *child nodes*. Nodes that have no children are called *leaf nodes*.

The `JTree` class can be used to display database metadata, both in terms of the complexity of the structures it can handle and in terms of the nature of the nodes in the structure. A `JTree` component is used for displaying any tree structure virtually. The nodes in a tree can be any type of object, and they all can be different if you need. To display the menu in a window, you can use the tree structure.

That's it for the overview of this chapter. You can see that there's a lot coming up. It's time to turn to the "Immediate Solutions" section.

Immediate Solutions

Handling Viewports

"Hmm," says the NP, "I want to move an image around under programmatic control, without displaying any scrollbars to the user. Is there any easy way to do that?" "As it happens," you say, "there is—you can use a viewport."

Viewports represent windows or portals into a view. For example, imagine that you have a huge document, not all of which will fit on the screen at one time. To present the user with only a part of that document at once, you can set up a viewport into that document, moving the viewport around the document until it fits. In fact, you can have multiple viewports to let the user move around inside the data in your program's model at will. Viewports are supported by the `JViewport` class, which is the foundation of scrolling in Swing. The inheritance diagram for this class is as follows:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.JViewport
```

You'll find the fields for the `JViewport` class in Table 13.1, its constructor in Table 13.2, and its methods in Table 13.3:

Table 13.1: Fields of the JViewport class

Field	Does this
`protected boolean backingStore`	It is deprecated and returns True when the viewport maintains an offscreen image of its contents
`protected Image backingStoreImage`	It is the image used for a backing store
`static int BACKINGSTORE_SCROLL_MODE`	It draws the viewport contents into an offscreen image
`static int BLIT_SCROLL_MODE`	It uses the `graphics.copyArea()` method to implement scrolling
`protected boolean isViewSizeSet`	It returns True when the viewport dimensions have been set
`protected Point lastPaintPosition`	It indicates the last `viewPosition` painted
`protected boolean scrollUnderway`	It indicates whether a scrolling operation is underway
`static int SIMPLE_SCROLL_MODE`	It uses a very simple method to redraw the entire contents of the scrollpane each time it is scrolled

Table 13.2: Constructor of the JViewport class

Constructor	Does this
`JViewport()`	It constructs a `JViewport` object

Table 13.3: Methods of the JViewport class

Method	Does this
`void addChangeListener (ChangeListener l)`	It adds a change listener
`protected void addImpl (Component child, Object constraints, int index)`	It sets the viewport's lightweight child, which can be null
`protected boolean computeBlit(int dx, int dy, Point blitFrom, Point blitTo, Dimension blitSize, Rectangle blitPaint)`	It computes the parameters for a blit operation

Table 13.3: Methods of the JViewport class

Method	Does this
`protected LayoutManager createLayoutManager()`	It installs a different layout manager
`protected JViewport.ViewListener createViewListener()`	It constructs a listener for the view
`protected void firePropertyChange(String propertyName, Object oldValue, Object newValue)`	It notifies listeners of a property change
`protected void fireStateChanged()`	It notifies listeners of a state change
`AccessibleContext getAccessibleContext()`	It gets the accessible context
`Dimension getExtentSize()`	It gets the size of the visible part of the view
`Insets getInsets()`	It gets the insets (border) dimensions as (0, 0, 0, 0) as the borders are not supported on a `JViewport` object
`Insets getInsets(Insets insets)`	It gets an `Insets` object containing this `JViewport` object's inset values
`int getScrollMode()`	It returns the current scrolling mode
`ViewportUI getUI()`	It returns the look and feel object that renders this component
`String getUIClassID()`	It returns a string that specifies the name of the look and feel class that renders this component
`Component getView()`	It gets the JViewport's one child or null
`Point getViewPosition()`	It gets the view coordinates that appear in the upper left corner of the viewport (0, 0 if there's no view)
`Rectangle getViewRect()`	It returns a rectangle whose origin is `getViewPosition` and size is `getExtentSize()`
`Dimension getViewSize()`	It gets the preferred size if the size has been set, otherwise this method returns the view's current size
`boolean isBackingStoreEnabled()`	It is deprecated and replaced by `getScrollMode()`
`boolean isOptimizedDrawingEnabled()`	`JViewport` overrides this method to return False
`void paint(Graphics g)`	It paints the image
`protected String paramString()`	It gets a string representation of this `JViewport` object
`void remove (Component child)`	It removes the viewport's lightweight child
`void removeChangeListener (ChangeListener l)`	It removes a change listener from the list
`void repaint(long tm, int x, int y, int w, int h)`	It repaints the list
`void reshape(int x, int y, int w, int h)`	It sets the bounds of this viewport
`void scrollRectToVisible (Rectangle contentRect)`	It scrolls the view so that the `Rectangle` within the view becomes visible
`void setBackingStoreEnabled (boolean x)`	It is deprecated and replaced by `setScrollMode()`
`void setBorder (Border border)`	It sets a border
`void setExtentSize (Dimension newExtent)`	It sets the size of the visible part of the view by using view coordinates
`void setScrollMode (int mode)`	It is used to control the method of scrolling the viewport's contents

Table 13.3: Methods of the JViewport class	
Method	**Does this**
`void setView(Component view)`	It sets the JViewport's lightweight child (its view)
`void setViewPosition (Point p)`	It sets the view coordinates that appear in the upper left corner of the viewport
`void setViewSize(Dimension newSize)`	It sets the view coordinates that appear in the upper left corner of the viewport as well as the size of the view
`Dimension toViewCoordinates (Dimension size)`	It converts a size in pixel coordinates to view coordinates
`Point toViewCoordinates (Point p)`	It converts a point in pixel coordinates to view coordinates
`void updateUI ()`	It resets the UI property to a value from the current look and feel

Let's take a look at a viewport example that enables the user to scroll the viewport, not with scrollbars but with buttons. Here, we add a viewport with an image in it to a program as well as add a panel containing buttons with the captions Scroll up and Scroll down. The user can scroll the image in the viewport by clicking the buttons.

We start by creating a new viewport and then a new panel. We add a label with an image in it to the panel and then add the panel to the viewport. When the user scrolls the viewport, he/she is actually scrolling the panel. The following code shows how we create the viewport and add it to the program (note that to actually add the panel to the viewport, we use the `JViewport` class's `setView()` method, which sets the viewport's object):

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
  CODE=DemoJViewport.class
  WIDTH=500
  HEIGHT=200 >
</APPLET>*/
public class DemoJViewport extends JApplet {
        public DemoJViewport() {
Container c = getContentPane();
JViewport port = new JViewport();
JPanel jp = new JPanel();
jp.add(new JLabel(new ImageIcon("illus.jpg")));
port.setView(jp);
c.add(port, BorderLayout.CENTER);
c.add(new ButtonPanel(port));
 }
  }
```

We also need another panel that's not a part of the viewport to display the buttons the user can use for scrolling. We'll call this panel `ButtonPanel`. Note that we must pass the viewport to this class's constructor so that it can scroll the viewport. Here's how we store the viewport passed to the constructor and add the buttons needed for scrolling:

```
class ButtonPanel extends JPanel implements ActionListener {
  JViewport port;
  JButton button1 = new JButton("Scroll up");
  JButton button2 = new JButton("Scroll down");

public ButtonPanel(JViewport vp) {
port = vp;
add(button1);
add(button2);
button1.addActionListener(this);
button2.addActionListener(this);
}
/* some code */
}
```

All that's left is to implement scrolling when the user clicks a button. To do that, we use the `JViewport` class's `getViewPosition()` method to get the current view position. Then, we change that position—according to the button clicked—by 15 pixels and use the `setViewPosition()` method to set the new viewport position, as shown in the following code:

```
    public void actionPerformed(ActionEvent e) {
    Point position = port.getViewPosition();
    if(e.getSource() == button1)
    position.y = +15;
    else if(e.getSource() == button2)
    position.y =-15;
    port.setViewPosition(position);
    }
    }
```

Now, we add a `ButtonPanel` object in the applet at the bottom as follows:

```
    public class DemoJViewport extends JApplet {
            public DemoJViewport() {
    Container c = getContentPane();
    JViewport port = new JViewport();
    JPanel jp = new JPanel();
    jp.add(new JLabel(new ImageIcon("illus.jpg")));
    port.setView(jp);
    c.add(port, BorderLayout.CENTER);
    c.add(new ButtonPanel(port), BorderLayout.SOUTH); }
    }
```

The result appears as shown in Figure 13.1:

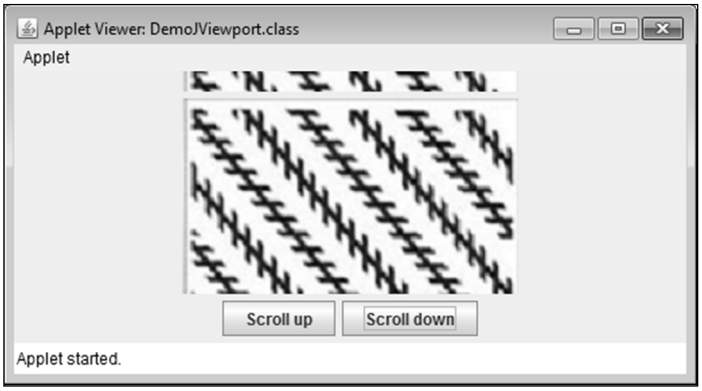


Figure 13.1: Scrolling a Viewport

You can see in the figure that the scroll buttons appear at the bottom of the applet. When the user clicks a button, the image scrolls accordingly.

Normally, you won't go all the way to the `JViewport` class to handle your scrolling, instead you'll use components, such as scroll panes, which are coming up next.

Creating Scroll Panes

The NP appears and says, "Hey, Java's gone all wacky again! We created a new Swing list control, put 40,000 items in it, made only two visible at once, and there are no scrollbars! This is definitely a bug, we are going to file a report and…" "Hold it," you say. "The `JList` control does not implement scrolling by itself, but you can put it into a scroll pane." The NP asks, "Really? Is that the way it's *supposed* to work?"

The `JScrollPane` class is the Swing lightweight implementation of a scroll pane, and you use it to scroll other controls. In fact, whenever you display a list, it's a standard to display it in a scroll pane to allow the user to scroll the list. Becouse of to the new Swing `Scrollable` interface, scroll operations are much more closely

coordinated with the control being scrolled (the `JViewport`, `JScrollPane`, and `JScrollBar` classes all implement this interface). The inheritance diagram for the `JScrollPane` class is as follows:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.JScrollPane
```

You'll find the fields for the `JScrollPane` class in Table 13.4, its constructors in Table 13.5, and its methods in Table 13.6:

Table 13.4: Fields of the JScrollPane class

Field	Does this
`protected JViewport columnHeader`	The column header child
`protected JScrollBar horizontalScrollBar`	The scroll pane's horizontal scrollbar child
`protected int horizontalScrollBarPolicy`	The display policy for the horizontal scrollbar
`protected Component lowerLeft`	The component to display in the lower left corner
`protected Component lowerRight`	The component to display in the lower right corner
`protected JViewport rowHeader`	The row header child
`protected Component upperLeft`	The component to display in the upper left corner
`protected Component upperRight`	The component to display in the upper right corner
`protected JScrollBar verticalScrollBar`	The scroll pane's vertical scrollbar child
`protected int verticalScrollBarPolicy`	The display policy for the vertical scrollbar
`protected JViewport viewport`	The scroll pane's viewport child

Table 13.5: Constructors of the JScrollPane class

Constructor	Does this
`JScrollPane()`	It constructs an empty `JScrollPane` object.
`JScrollPane(Component view)`	It constructs a `JScrollPane` object that displays the contents of the indicated component.
`JScrollPane(Component view, int vsbPolicy, int hsbPolicy)`	It constructs a `JScrollPane` object that displays the view component in a viewport whose view position can be controlled with a pair of scrollbars.
`JScrollPane(int vsbPolicy, int hsbPolicy)`	It constructs an empty `JScrollPane` object with indicated scrollbar policies.

Table 13.6: Methods of the JScrollPane class

Method	Does this
`JScrollBar createHorizontalScrollBar()`	It creates the horizontal scrollbar.
`JScrollBar createVerticalScrollBar()`	It creates the vertical scrollbar.
`protected JViewport createViewport()`	It gets a new `JViewport` object by default.
`AccessibleContext getAccessibleContext()`	It gets the accessible context associated with this `JScrollPane` object.
`JViewport getColumnHeader()`	It gets the column header.
`Component getCorner(String key)`	It gets the component at the indicated corner.
`JScrollBar getHorizontalScrollBar()`	It gets the horizontal scrollbar.
`int getHorizontalScrollBarPolicy()`	It gets the horizontal scrollbar policy value.
`JViewport getRowHeader()`	It gets the row header.

Table 13.6: Methods of the JScrollPane class

Method	Does this
`ScrollPaneUI getUI()`	It gets the look and feel object that renders this component.
`String getUIClassID()`	It gets the key used to look up the `ScrollPaneUI` class that provides the look and feel for `JScrollPane`.
`JScrollBar getVerticalScrollBar()`	It gets the vertical scrollbar.
`int getVerticalScrollBarPolicy()`	It gets the vertical scrollbar policy value.
`JViewport getViewport()`	It gets the current `JViewport` object.
`Border getViewportBorder()`	It gets the value of the `Border` object that surrounds the viewport.
`Rectangle getViewportBorderBounds()`	It gets the bounds of the viewport border.
`boolean isValidateRoot()`	It checks the root.
`protected String paramString()`	It gets a string representation of this `JScrollPane` object.
`void setColumnHeader (JViewport columnHeader)`	If an old column header exists, this method removes it. If the new column header is not null, it synchronizes the x coordinates of its view position with the viewport (if there is one) and then adds it to the scroll pane.
`void setColumnHeaderView (Component view)`	It constructs a column-header viewport, if necessary, sets its view, and then adds the column-header viewport to the scroll pane.
`void setCorner(String key, Component corner)`	It adds a child that will appear in one of the scroll pane's corners (if there's room).
`void setHorizontalScrollBar (JScrollBar horizontalScrollBar)`	It adds the scrollbar that controls the viewport's horizontal view position to the scroll pane.
`void setHorizontalScrollBarPolicy (int policy)`	It determines when the horizontal scrollbar appears in the scroll pane.
`void setLayout (LayoutManager layout)`	It sets the layout manager for this `JScrollPane` object.
`void setRowHeader (JViewport rowHeader)`	It sets the new row header. If the new `rowHeader` is not null, it synchronizes the y coordinates of its view position with the viewport (if there is one) and then adds it to the scroll pane.
`void setRowHeaderView (Component view)`	It constructs a row-header viewport, if necessary, and then adds the row-header viewport to the scroll pane.
`void setUI (ScrollPaneUI ui)`	It sets the object that provides the look and feel.
`void setVerticalScrollBar (JScrollBar verticalScrollBar)`	It adds the scrollbar that controls the viewport's vertical view position.
`void setVerticalScrollBarPolicy(int policy)`	It determines when the vertical scrollbar appears in the scroll pane.
`void setViewport(JViewport viewport)`	It forces the view position of the new viewport to be in the +x, +y quadrant.
`void setViewportBorder (Border viewportBorder)`	It adds a border around the viewport.
`void setViewportView (Component view)`	It constructs a viewport, if necessary, and then sets its view.
`void updateUI()`	It is called when the default look and feel changes.

Let's look at an example in which we add a grid layout of text fields to a panel and scroll that panel in a scroll pane. We'll start by creating a `JPanel` object full of text fields as follows:

```
import java.awt.*;
import javax.swing.*;
/*<APPLET
  CODE=DemoJScrollPane.class
  WIDTH=300
  HEIGHT=200 >
</APPLET>*/
public class DemoJScrollPane extends JApplet {
         public void init() {
  Container c = getContentPane();
  JPanel jp = new JPanel();
  jp.setLayout(new GridLayout(11, 16));
  for(int outer = 0; outer <= 10; outer++) {
  for(int inner = 0; inner <= 15; inner++) {
  jp.add(new JTextField("Text Field " + outer +
  ", " + inner)); }
  }
  /* some code */
}
}
```

Now, we'll add this new panel to a scroll pane. When you create a scroll pane object, you pass the object to be scrolled, and you can specify when and where you want the scrollbars with these constants:

- HORIZONTAL_SCROLLBAR_ALWAYS
- HORIZONTAL_SCROLLBAR_AS_NEEDED
- HORIZONTAL_SCROLLBAR_NEVER
- VERTICAL_SCROLLBAR
- VERTICAL_SCROLLBAR_ALWAYS
- VERTICAL_SCROLLBAR_AS_NEEDED
- VERTICAL_SCROLLBAR_NEVER

In this case, we'll always let the scroll pane display scrollbars, as shown in the following code:

```
import java.awt.*;
import javax.swing.*;
/*<APPLET
  CODE=DemoJScrollPane.class
  WIDTH=300
  HEIGHT=200 >
</APPLET>*/
public class DemoJScrollPane extends JApplet {
         public void init() {
Container c = getContentPane();
JPanel jp = new JPanel();
jp.setLayout(new GridLayout(10, 15));
for(int outer = 0; outer <= 9; outer++) {
         for(int inner = 0; inner <= 14; inner++) {
         jp.add(new JTextField("Text Field " + outer +
         ", " + inner)); }
         }
  JScrollPane scrollpane = new JScrollPane(jp,
  ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
  ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS);
  c.add(scrollpane); }
}
```

The result of adding the scroll pane to the applet's content pane is shown in Figure 13.2:

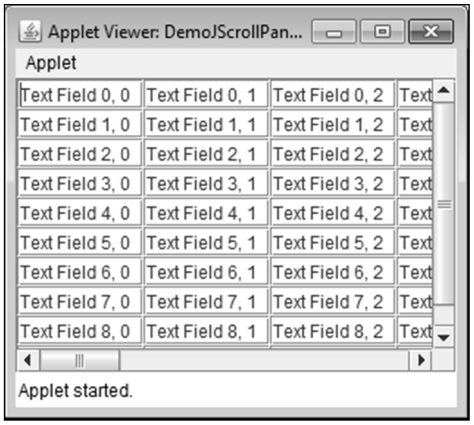


Figure 13.2: Using a Scroll Pane

You see in the figure that the whole grid of text fields appears in the scroll pane, and the user can scroll that grid with the scrollbars.

Creating Scroll Pane Headers and Borders

The Programming Correctness Czar (PCC) appears and says, "So, Java has scroll panes? Well, they don't seem to be very professional—with professional controls, you could specify labels or even images to use as row and column headers." "No problem," you say. "You can do that in Java, and you can even select the border type as well." "Oh," says the PCC.

You can customize a scroll pane by adding header images or text with the `setColumnHeaderView()` and `setRowHeaderView()` methods. You can also customize the border used in a scroll pane with the `setViewportBorder()` method.

TIP

You can customize the corners of a scroll pane with the setCorner() method.

As an example, we'll put this customization to work in the scroll pane example that we have developed in the previous solution. In this case, we'll use labels as the row and column headers and add an etched border, as shown in the following code:

```
import java.awt.*;
import javax.swing.*;
/*<APPLET
  CODE=DemoJScrollPane1.class
  WIDTH=300
  HEIGHT=200 >
</APPLET>*/
public class DemoJScrollPane1 extends JApplet {
        public void init() {
  Container c = getContentPane();
  c.setLayout(new BorderLayout());
  JPanel jp = new JPanel();
  jp.setLayout(new GridLayout(10, 15));
  for(int outer = 0; outer <= 9; outer++) {
  for(int inner = 0; inner <= 14; inner++) {
  jp.add(new JTextField("Text Field " + outer +
  ", " + inner)); }
  }
  JScrollPane scrollpane = new JScrollPane(jp,
  ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
```

```
      ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS);
      JLabel jl1 = new JLabel("Horizontal label");
      JLabel jl2 = new JLabel("Vertical label");
      scrollpane.setColumnHeaderView(jl1);
      scrollpane.setRowHeaderView(jl2);
      scrollpane.setViewportBorder(BorderFactory.createEtchedBorder());
      c.add(scrollpane); }
   }
```

You can see the result of this code in Figure 13.3:

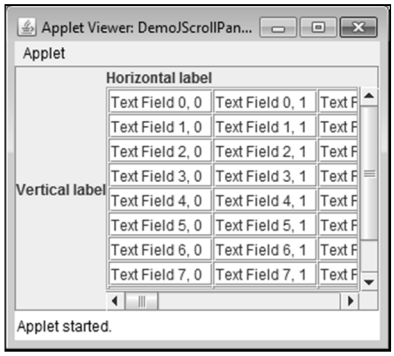


Figure 13.3: Adding Headers to a Scroll Pane

Scrolling Images

Scroll panes provide an ideal way to scroll images—all you have to do is display the image in a label (or a similar component) and add that label to a scroll pane. Here's an example showing how this works:

```
import java.awt.*;
import javax.swing.*;
/*<APPLET
   CODE=DemoJScrollPane2.class
   WIDTH=300
   HEIGHT=200 >
</APPLET>*/
public class DemoJScrollPane2 extends JApplet {
        public DemoJScrollPane2() {
        Container c = getContentPane();
        JLabel jl = new JLabel(new ImageIcon("bulg.jpg"));
        JScrollPane scrollpane = new JScrollPane(jl);
        c.add(scrollpane); }
}
```

The result appears as shown in Figure 13.4:

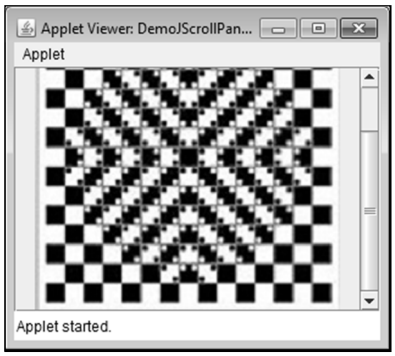


Figure 13.4: Scrolling an Image

Now, the user can scroll a large image, as needed, to handle an image that can't be displayed all at once.

Creating Sliders

The NP is back and says, "I have seen a new control in various programs—sliders. When do you think will we get those in Java?" You smile and say, "Quite a while ago."

You can use scrollbars to let the users select a value from a continuous range, but users have become more familiar with using scrollbars to scroll other controls, such as lists. Therefore, Swing contains the `JSlider` class to support slider controls.

Sliders present the user with controls that look like those you see in audio equipment; they are clearly intended to let the user select a value from a range. To use the slider, the user moves a knob around with the mouse. The inheritance diagram for the `JSlider` class is as follows:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.JSlider
```

You can find the fields for the `JSlider` class in Table 13.7, its constructors in Table 13.8, and its methods in Table 13.9:

Table 13.7: Fields of the JSlider class

Field	Does this
`protected ChangeEvent changeEvent`	The change event
`protected ChangeListener changeListener`	The change listener
`protected int majorTickSpacing`	The number of values between the major tick marks
`protected int minorTickSpacing`	The number of values between the minor tick marks
`protected int orientation`	The orientation of the slider
`protected BoundedRangeModel sliderModel`	The data model, which handles the numeric maximum value, minimum value, and current position value for the slider
`protected boolean snapToTicks`	If True, the thumb resolves to the closest tick mark next to where the user positioned the thumb

Table 13.8: Constructors of the JSlider class

Constructor	Does this
`JSlider()`	It constructs a horizontal slider with the range from 0 to 100 and an initial value of 50
`JSlider(BoundedRangeModel brm)`	It constructs a horizontal slider by using the indicated bounded range model
`JSlider(int orientation)`	It constructs a slider by using the indicated orientation with the range from 0 to 100 and an initial value of 50
`JSlider(int min, int max)`	It constructs a horizontal slider by using the indicated minimum and maximum values with an initial value of 50
`JSlider(int min, int max, int value)`	It constructs a horizontal slider by using the indicated minimum, maximum, and initial values
`JSlider(int orientation, int min, int max, int value)`	It constructs a slider with the indicated orientation and the indicated minimum, maximum, and initial values

Table 13.9: Methods of the JSlider class

Method	Does this
`void addChangeListener (ChangeListener l)`	It adds a change listener to the slider

Table 13.9: Methods of the JSlider class

Method	Does this
`protected ChangeListener createChangeListener()`	You override this method to return your own change listener implementation
`Hashtable createStandardLabels(int increment)`	It constructs a hashtable that will draw text labels starting at the slider minimum
`Hashtable createStandardLabels(int increment, int start)`	It constructs a hashtable that will draw text labels starting at the start point indicated
`protected void fireStateChanged()`	It sends a change event, whose source is this slider, to each listener
`AccessibleContext getAccessibleContext()`	It gets the accessible context
`int getExtent()`	It gets the extent, which is the range of values covered by the thumb
`boolean getInverted()`	It returns True if the value range shown for the slider is reversed
`Dictionary getLabelTable()`	It gets the dictionary of what labels to draw
`int getMajorTickSpacing()`	It returns the major tick spacing
`int getMaximum()`	It gets the maximum value supported by the slider
`int getMinimum()`	It gets the minimum value supported by the slider
`int getMinorTickSpacing()`	It returns the minor tick spacing
`BoundedRangeModel getModel()`	It gets the data model that handles the slider's three fundamental properties: minimum, maximum, and value
`int getOrientation()`	It returns this slider's vertical or horizontal orientation
`boolean getPaintLabels()`	It indicates whether labels are to be painted
`boolean getPaintTicks()`	It indicates whether tick marks are to be painted
`boolean getPaintTrack()`	It indicates whether the track is to be painted
`boolean getSnapToTicks()`	It returns True if the thumb resolves to the closest tick mark next to where the user positioned the thumb
`SliderUI getUI()`	It gets the UI object, which implements the look and feel for this component
`String getUIClassID()`	It gets the name of the look and feel class that renders this component
`int getValue()`	It gets the slider's value
`boolean getValueIsAdjusting()`	It returns True if the slider thumb is being dragged
`protected String paramString()`	It gets a string representation of this `JSlider` object
`void removeChangeListener (ChangeListener l)`	It removes a change listener from the slider
`void setExtent(int extent)`	It sets the size of the range covered by the thumb
`void setFont(Font)`	It sets the font for this component
`void setInverted (boolean b)`	It passes a value of True to reverse the value range
`void setLabelTable (Dictionary labels)`	It is used to specify what label will be drawn at any given value
`void setMajorTickSpacing (int n)`	It sets the major tick spacing
`void setMaximum (int maximum)`	It sets the model's maximum property
`void setMinimum (int minimum)`	It sets the model's minimum property
`void setMinorTickSpacing (int n)`	It sets the minor tick spacing

Table 13.9: Methods of the JSlider class	
Method	**Does this**
`void setModel (BoundedRangeModel newModel)`	It sets the model that handles the slider's three fundamental properties: minimum, maximum, and value
`void setOrientation(int orientation)`	It sets the scrollbar's orientation to either vertical or horizontal
`void setPaintLabels (boolean b)`	It determines whether labels are painted on the slider
`void setPaintTicks (boolean b)`	It determines whether tick marks are painted on the slider
`void setPaintTrack (boolean b)`	It determines whether the track is painted on the slider
`void setSnapToTicks (boolean b)`	It returns True to make the thumb resolve to the closest tick mark next to where the user positioned the thumb
`void setUI(SliderUI ui)`	It sets the UI object, which implements the look and feel for this component
`void setValue(int n)`	It sets the slider's current value
`void setValueIsAdjusting (boolean b)`	It sets the model's `ValueIsAdjusting` property
`protected void updateLabelUIs()`	It replaces the label UIs with the latest versions from the `UIFactory` class
`void updateUI()`	It changes the look and feel of the user interface

Let's put the `JSlider` class to work in an example. In this case, we'll just create a slider that can return values from 0 to 80 and report the current value in an applet's status bar. The following code shows how we create a horizontal slider with the constant `SwingConstants.HORIZONTAL` (as you might guess, the other possibility is `SwingConstants.VERTICAL`) with a minimum value of 0, maximum value of 80, and initial value of 0:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;
/* <APPLET
  CODE=DemoJSlider.class
  WIDTH=300
  HEIGHT=200 >
</APPLET>
*/
public class DemoJSlider extends JApplet implements ChangeListener {
        JSlider slider = new JSlider(SwingConstants.HORIZONTAL, 0, 80, 0);
public void init() {
/* some code */
}
}
```

Now, in the `init()` method, we'll add this slider to the applet's content pane. You use *change listeners* with sliders, not adjustment listeners, as you do with scrollbars. The `ChangeListener` interface has only one method, that is, `stateChanged()`, which you can see in Table 13.10:

Table 13.10: Method of the ChangeListener interface	
Method	**Does this**
`void stateChanged (ChangeEvent e)`	It is invoked when the target of the listener has changed its state

We add a change listener to the slider in this applet and then add the slider to the applet's layout, as shown in the following code:

```
public void init() {
  Container c = getContentPane();
  c.setLayout(new FlowLayout());
  slider.addChangeListener(this);
  c.add(slider);
}
```

In the `stateChanged()` method, we use the `JSlider()` methods `getMinimum()`, `getMaximum()`, and `getValue()` to display the settings for the slider in the status bar, as shown in the following code:

```
public void stateChanged(ChangeEvent e) {
  JSlider js = (JSlider) e.getSource();
  showStatus("Slider minimum: " + js.getMinimum() +
  ", maximum: " + js.getMaximum() +
  ", value: " + js.getValue());
}
```

The result appears as shown in Figure 13.5:

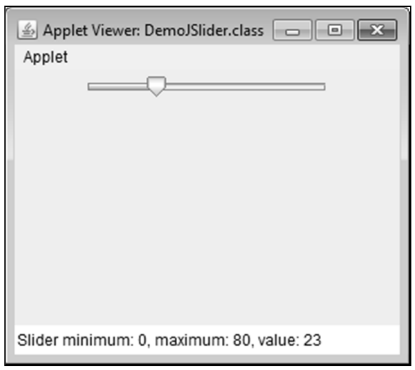


Figure 13.5: Using a Slider

You can see in the figure that the user can move the knob on the slider, and the new value appears.

Filling a Slider

If you are using the default Metal look in Swing, you can *fill in* a slider, which means that its track will appear filled from the origin to the slider's knob. To do that, you must set the `JSlider` class's `isFilled` *client property* to True, which you can do as follows (note that we are adding this code to the slider example from the previous solution):

```
public void init() {
  Container c = getContentPane();
  c.setLayout(new FlowLayout());
  slider.addChangeListener(this);
  slider.putClientProperty("JSlider.isFilled", Boolean.TRUE);
  c.add(slider);
}
```

The result appears as shown in Figure 13.6:

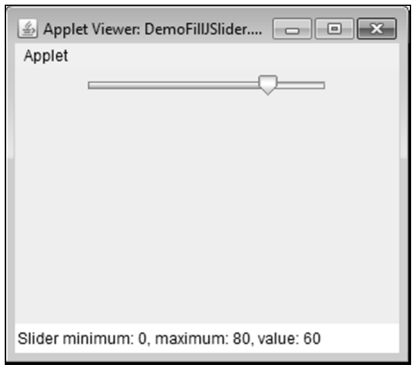


Figure 13.6: Filling a Slider

You can see that the slider in the figure is filled. The output of both the figures, Figure 13.5 and Figure 13.6, appears same; therefore, the `JSlider` is filled by default. Try to run `DemoFillJSlider.java` by passing `Boolean.FALSE` in the method `putClientProperty()` to notice the difference.

Painting Slider Tick Marks

"Hmm," says the NP, "users are complaining about the sliders in my program—they say they can't get their bearings by using a single long slider." "Well," you say, "you can fix that by adding tick marks."

To paint tick marks in a slider control, you pass a value of True to the `setPaintTicks()` method and then indicate the spacing you want for the major (that is, *longer*) and minor (*shorter*) ticks by using the `setMajorTickSpacing()` and `setMinorTickSpacing()` methods, as shown in the following code (note that we are adding this code to the existing slider example):

```
public void init() {
  Container c = getContentPane();
  c.setLayout(new FlowLayout());
  slider.addChangeListener(this);
  slider.setPaintTicks(true);
  slider.setMajorTickSpacing(20);
  slider.setMinorTickSpacing(10);
  c.add(slider);
}
```

The result appears as shown in Figure 13.7:

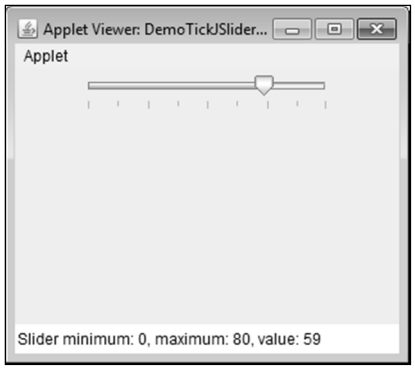


Figure 13.7: Painting Tick Marks in a Slider

You can see in the figure that the slider now displays tick marks. You can also paint the numeric values for the major tick marks—see the next solution for the details.

Painting Slider Labels

You can use the `JSlider` class's `setPaintLabels()` method to display the numeric value for the major tick marks in the slider. Here's how we do that by adding one line of code to the existing slider example:

```
public void init() {
  Container c= getContentPane();
  c.setLayout(new FlowLayout());
  slider.addChangeListener(this);
  slider.putClientProperty("JSlider.isFilled", Boolean.TRUE);
  slider.setPaintTicks(true);
  slider.setPaintLabels(true);
  slider.setMajorTickSpacing(20);
  slider.setMinorTickSpacing(10);
  c.add(slider);
}
```

The result appears as shown in Figure 13.8:

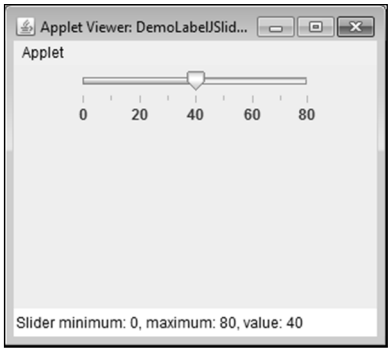


Figure 13.8: Painting Slider Labels

You can see in the figure that the major tick marks are labeled.

Setting the Slider Extent

You can set a slider's *extent* that bounds the maximum value in the slider. If you set the extent of a slider and its maximum value is *maximum*, the slider's value can never exceed *maximum* minus *extent*. The following code is an example in which we'll add a button to the slider example we have been developing over the last few solutions, which lets the user set the slider's extent to 50:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;
/*<APPLET
  CODE=DemoExtentSlider.class
  WIDTH=400
  HEIGHT=200 >
</APPLET>*/
public class DemoExtentSlider extends JApplet implements ActionListener,
ChangeListener {
        JSlider slider = new JSlider(SwingConstants.HORIZONTAL, 0, 80, 0);
        JButton button = new JButton("Set extent to 50");
        public void init() {
  Container c = getContentPane();
  c.setLayout(new FlowLayout());
  slider.addChangeListener(this);
  slider.putClientProperty("JSlider.isFilled", Boolean.TRUE);
  slider.setPaintTicks(true);
  slider.setPaintLabels(true);
  slider.setMajorTickSpacing(20);
  slider.setMinorTickSpacing(10);
  slider.revalidate();
  c.add(slider);
  button.addActionListener(this);
  c.add(button); }
        public void stateChanged(ChangeEvent e) {
  JSlider js = (JSlider) e.getSource();
  showStatus("Slider minimum: " + js.getMinimum() +
  ", maximum: " + js.getMaximum() +
  ", value: " + js.getValue() +
  ", extent: " + js.getExtent()); }
        public void actionPerformed(ActionEvent e) {
  slider.setValue(30);
  slider.setExtent(50);
  slider.revalidate(); }
}
```

When the user clicks the button, the slider's extent is set to 50, and as its maximum value is 80, the slider's value can't exceed 30. You can see the result in Figure 13.9:

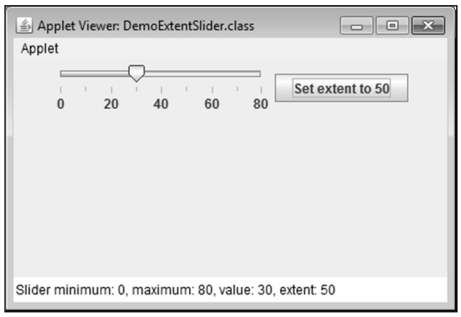


Figure 13.9: Setting a Slider's Extent

Note that even though the slider is at its maximum setting, its value is only 30.

Creating Scrollbars

"OK," says the NP, "I want to let the users move through the data in my program. I want to let them adjust values. I want to let them navigate through the items in a long list. I want...." "Scrollbars," you say. "What you want is scrollbars." "Right," says the NP.

The Swing lightweight scrollbar class is `JScrollBar`, and its inheritance diagram is as follows:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.JScrollBar
```

You can find the fields of the `JScrollBar` class in Table 13.11, its constructors in Table 13.12, and its methods in Table 13.13:

Table 13.11: Fields of the JScrollBar class

Field	Does this
`protected int blockIncrement`	The block increment
`protected BoundedRangeModel model`	The model that represents the scrollbar's minimum, maximum, extent, and current values
`protected int orientation`	The scrollbar's orientation
`protected int unitIncrement`	The unit increment

Table 13.12: Constructors of the JScrollBar class

Constructor	Does this
`JScrollBar()`	It constructs a vertical scrollbar
`JScrollBar(int orientation)`	It constructs a scrollbar with the indicated orientation and provides the values
`JScrollBar(int orientation, int value, int extent, int min, int max)`	It constructs a scrollbar with the indicated orientation, value, extent, minimum, and maximum values

Table 13.13: Methods of the JScrollBar class

Method	Does this
`void addAdjustmentListener(AdjustmentListener l)`	It adds an adjustment listener

Table 13.13: Methods of the JScrollBar class

Method	Does this
`protected void fireAdjustmentValueChanged(int id, int type, int value)`	It fires an adjustment event
`AccessibleContext getAccessibleContext()`	It gets the accessible context
`int getBlockIncrement()`	It is used for backward compatibility with `java.awt.Scrollbar` only
`int getBlockIncrement(int direction)`	It gets the amount to change the scrollbar's value by the given value, given a block change request
`int getMaximum()`	It gets the maximum value of the scrollbar that equals the maximum value minus the extent
`Dimension getMaximumSize()`	It gets the scrollbar's maximum size
`int getMinimum()`	It gets the minimum value supported by the scrollbar
`Dimension getMinimumSize()`	It gets the scrollbar's minimum size
`BoundedRangeModel getModel()`	It gets the data model that handles the scrollbar's four fundamental properties: minimum, maximum, value, and extent
`int getOrientation()`	It gets the component's orientation (horizontal or vertical)
`ScrollBarUI getUI()`	It gets the delegate that implements the look and feel for this component
`String getUIClassID()`	It gets the name of the look and feel class for this component
`int getUnitIncrement()`	It is used for backward compatibility with `java.awt.Scrollbar` only
`int getUnitIncrement (int direction)`	It gets the amount to change the scrollbar's value by the given value, given a unit change request
`int getValue()`	It gets the scrollbar's value
`boolean getValueIsAdjusting()`	It returns True if the scrollbar knob is being dragged
`int getVisibleAmount()`	It gets the scrollbar's extent
`boolean isFocusTraversable ()`	It identifies whether this component can receive the focus
`protected String paramString()`	It gets a string representation of the scrollbar
`void removeAdjustmentListener (AdjustmentListener l)`	It removes an `AdjustmentEvent` listener
`void setBlockIncrement(int blockIncrement)`	It sets the `blockIncrement` property
`void setEnabled(Boolean x)`	It enables the component so that the knob position can be changed
`void setMaximum(int maximum)`	It sets the model's maximum property
`void setMinimum(int minimum)`	It sets the model's minimum property
`void setModel (BoundedRangeModel newModel)`	It sets the model that handles the scrollbar's four fundamental properties: minimum, maximum, value, and extent
`void setOrientation (int orientation)`	It sets the scrollbar's orientation to either vertical or horizontal
`void setUnitIncrement (int unitIncrement)`	It sets the `unitIncrement` property
`void setValue(int value)`	It sets the scrollbar's value

Table 13.13: Methods of the JScrollBar class	
Method	**Does this**
`void setValueIsAdjusting (boolean b)`	It sets the model's `valueIsAdjusting` property
`void setValues(int newValue, int newExtent, int newMin, int newMax)`	It sets the four `BoundedRangeModel` properties
`void setVisibleAmount (int extent)`	It sets the model's extent property
`void updateUI()`	It overrides `JComponent.updateUI`

Let's take an example in which we use a scrollbar to scroll some text in an applet. To do that, we create a new panel that displays a label with the text "Hello ScrollBar!" at the vertical position given by the public data member named `vertical`. Here's what the panel class looks like:

```
class DemoPanel extends JPanel {
         JLabel label = new JLabel("Hello ScrollBar!");
         int vertical = 0;
DemoPanel() { label = new JLabel("Hello ScrollBar!"); add(label); }
         public void paintComponent(Graphics g) {
   super.paintComponent(g);
   label.setLocation(0, vertical); }
public void setScrolledPosition(int newposition) { vertical = newposition; }
}
```

Now, we can add an object of this new panel class to an applet and a vertical scrollbar in a border layout as follows:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
   CODE=DemoJScrollBar.class
   WIDTH=300
   HEIGHT=200 >
</APPLET>*/
public class DemoJScrollBar extends JApplet {
         private JScrollBar sbar = new JScrollBar(JScrollBar.VERTICAL, 0,
         0, 0, 150);
         private DemoPanel jp = new DemoPanel();
         public void init() {
         Container c = getContentPane();
         c.add(jp, BorderLayout.CENTER);
         c.add(sbar, BorderLayout.EAST);
         sbar.addAdjustmentListener(new AdjustmentListener() {
         public void adjustmentValueChanged(
         AdjustmentEvent e) {
         JScrollBar sb = (JScrollBar)e.getSource();
         jp.setScrolledPosition(e.getValue());
         jp.repaint(); }});
   }
}
```

The result appears as shown in Figure 13.10:

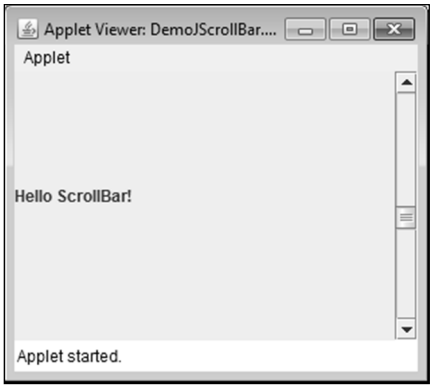


Figure 13.10: Creating and Using a Scrollbar

You can see in the figure that the user can use the scrollbar to move the text up and down in the panel.

Creating Lists

The Big Boss appears in a cloud of cigar smoke and says, "We have got 14,389 products now for the user to select from. How are we going to display all those in a program?" "No problem," you say. "I'll just use a list control." The Big Boss smiles and says, "Here are the product names to type in."

The `JList` class is the Swing lightweight list control. The inheritance diagram for the `JList` class is as follows:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.Jlist<E>
```

You can do more with `JList` controls than you can with AWT `List` controls; for example, you can display images in Swing lists. In fact, we'll take a look at what you can do with Swing lists from this point to the end of the chapter. You can find the constructors of the `JList` class in Table 13.14 and its methods in Table 13.15:

Table 13.14: Constructors of the JList class

Constructor	Does this
`JList()`	It constructs a `JList` object.
`JList(ListModel<E> dataModel)`	It constructs a `JList` object that displays the elements in the indicated data model.
`JList(E[ ] listData)`	It constructs a `JList` object that displays the elements in the indicated array.
`JList(Vector<? extends E> listData)`	It constructs a `JList` object that displays the elements in the indicated vector.

Table 13.15: Methods of the JList class

Method	Does this
`void addListSelectionListener (ListSelectionListener listener)`	It adds a list selection listener.
`void addSelectionInterval(int anchor, int lead)`	It makes the selection to be the union of the indicated interval with the current selection.
`void clearSelection()`	It clears the selection.
`protected ListSelectionModel createSelectionModel()`	It gets an instance of `DefaultListSelectionModel`.

Table 13.15: Methods of the JList class

Method	Does this
`void ensureIndexIsVisible(int index)`	It scrolls the viewport to make sure an item is visible.
`protected void fireSelectionValueChanged(int firstIndex, int lastIndex, boolean isAdjusting)`	It notifies the `JList` list selection listeners that the selection model has changed.
`AccessibleContext getAccessibleContext()`	It gets the accessible context.
`int getAnchorSelectionIndex()`	It gets the first index argument from the most recent `addSelectionInterval` or `setSelectionInterval` call.
`Rectangle getCellBounds(int index1, int index2)`	It gets the bounds of the indicated range of items.
`ListCellRenderer getCellRenderer()`	It gets the object that renders the list items.
`JList.DropLocation get DropLocation()`	It returns the location that this component should visually indicate as the drop location.
`DropMode getDropMode()`	It returns the drop mode for this component.
`int getFirstVisibleIndex()`	It returns the index of the cell in the upper left corner of the `JList` object.
`int getFixedCellHeight()`	It gets the fixed cell height value.
`int getFixedCellWidth()`	It gets the fixed cell width value.
`int getLastVisibleIndex()`	It returns the largest list index that is currently visible.
`int getLeadSelectionIndex()`	It gets the second index argument from the most recent `addSelectionInterval` or `setSelectionInterval` call.
`int getMaxSelectionIndex()`	It gets the maximum selected cell index.
`int getMinSelectionIndex()`	It gets the minimum selected cell index.
`ListModel getModel()`	It gets the data model that holds the list of data items.
`Dimension getPreferredScrollableViewportSize( )`	It computes the size of the viewport needed to display `VisibleRowCount` rows.
`Object getPrototypeCellValue()`	It gets the cell width of the prototypical cell (a cell used for the calculation of cell widths).
`int getScrollableBlockIncrement(Rectang le visibleRect, int orientation, int direction)`	It gets the block increment amount.
`boolean getScrollable tracksViewportHeight()`	It gets the height of the track's viewport.
`boolean getScrollable tracksViewportWidth()`	It gets the width of the track's viewport.
`int getScrollableUnitIncrement(Rectangl e visibleRect, int orientation, int direction)`	It returns the list's font size.
`int getSelectedIndex()`	It returns the first selected index.
`int[ ] getSelectedIndices()`	It returns an array of all the selected indexes in an increasing order.
`Object getSelectedValue()`	It returns the first selected value or null if the selection is empty.
`Object[] getSelectedValues()`	It returns an array of the values for the selected cells.
`Color getSelectionBackground()`	It gets the background color for selected cells.

Table 13.15: Methods of the JList class

Method	Does this
`Color getSelectionForeground()`	It gets the foreground color.
`int getSelectionMode()`	It gets whether single-item or multiple-item selections are allowed.
`ListSelectionModel getSelectionModel()`	It gets the value of the current selection model.
`ListUI getUI()`	It gets the look and feel object that renders this component.
`String getUIClassID()`	It gets the name of the `UIFactory` class that generates the look and feel for this component.
`boolean getValueIsAdjusting()`	It gets the value of the data model's `isAdjusting` property.
`int getVisibleRowCount()`	It returns the preferred number of visible rows.
`Point indexToLocation(int index)`	It gets the origin of the indicated item in `JList` coordinates. It gets null if the index isn't valid.
`boolean isSelectedIndex(int index)`	It returns True if the indicated index is selected.
`boolean isSelectionEmpty()`	It returns True if nothing is selected. This is a convenience method that just delegates to the selection model.
`int locationToIndex(Point location)`	It converts a point in `JList` coordinates to the index of the cell at that location.
`protected String paramString()`	It gets a string representation of this `JList` object.
`void removeListSelectionListener(ListSelectionListener listener)`	It removes a listener from the list that's notified each time a change to the selection occurs.
`void removeSelectionInterval (int index0, int index1)`	It sets the selection to be the set difference of the indicated interval and the current selection.
`void setCellRenderer (ListCellRenderer cellRenderer)`	It sets the delegate that's used to paint each cell in the list.
`void setDropMode(DropMode)`	It sets the drop mode for this component.
`void setFixedCellHeight (int height)`	It defines the height of every cell in the list.
`void setFixedCellWidth(int width)`	It defines the width of every cell in the list.
`void setListData(Object[ ] listData)`	It constructs a list model from an array of objects and then applies the `setModel()` method to it.
`void setListData(Vector listData)`	It constructs a list model from a vector and then applies the `setModel()` method to it.
`void setModel(ListModel model)`	It sets the model that represents the contents of the list and clears the list selection.
`void setPrototypeCellValue (Object prototypeCellValue)`	It sets the `prototypeCellValue` property and then computes the `fixedCellWidth` and `fixedCellHeight` properties.
`void setSelectedIndex(int index)`	It selects a single cell.
`void setSelectedIndices (int[ ] indices)`	It selects a set of cells.
`void setSelectedValue (Object anObject, boolean shouldScroll)`	It selects the indicated object from the list.
`void setSelectionBackground(Color selectionBackground)`	It sets the background color for the selected cells.
`void setSelectionForeground(Color selectionForeground)`	It sets the foreground color for the selected cells.

Table 13.15: Methods of the JList class	
Method	**Does this**
`void setSelectionInterval (int anchor, int lead)`	It selects the indicated interval.
`void setSelectionMode (int selectionMode)`	It determines whether single-item or multiple-item selections are allowed.
`void setSelectionModel (ListSelectionModel selectionModel)`	It sets the selection model for the list to a non-null `ListSelectionModel` implementation.
`void setUI(ListUI ui)`	It sets the look and feel object that renders this component.
`void setValueIsAdjusting (boolean b)`	It sets the data model's `isAdjusting` property to True.
`void setVisibleRowCount (int visibleRowCount)`	It sets the preferred number of rows in the list that can be displayed without a scrollbar.
`void updateUI()`	It sets the UI property with the `ListUI` object from the `DefaultUIFactory` class.

Note that you can get the currently selected item's index with the `JList` class's `getSelectedIndex()` method, multiple selections with the `getSelectedIndices()` method, the actual selected item with the `getSelectedValue()` method, and the actual selected items with the `getSelectedValues()` method.

TIP

You should keep in mind that the data in a list is actually maintained by its model, so if you want access to an item that's not selected, you can use the JList class's getModel() method to get the model and then the model's getElementAt() method to get the actual item.

We'll put the `JList` class to work in this and the following few solutions. To start, we create a simple `JList` example that just displays 10 items and reports which one the user clicks. We start by creating a list control with those items in it as follows:

```
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
/*<APPLET
  CODE=DemoJList.class
  WIDTH=300
  HEIGHT=200 >
</APPLET>*/
public class DemoJList extends JApplet implements ListSelectionListener {
  JList list;
  public void init() {
  Container c = getContentPane();
  String[] items = new String[10];
  for(int loop_index = 0; loop_index <= 9; loop_index++) {
  items[loop_index] = "Item " + loop_index; }
  list = new JList(items);
/* some code */   }
  /* some code  */
  }
```

Next, we add this list control to a scroll pane to enable scrolling, set it so that it will display four rows, and add a selection listener to it, as shown in the following code:

```
public void init() {
  Container c = getContentPane();
  String[] items = new String[10];
  for(int loop_index = 0; loop_index <= 9; loop_index++) {
        items[loop_index] = "Item " + loop_index; }
  list = new JList(items);
  JScrollPane scrollpane = new JScrollPane(list);
  list.setVisibleRowCount(4);
```

```
      list.addListSelectionListener(this);
      c.setLayout(new FlowLayout());
      c.add(scrollpane);
   }
```

Now, in the `valueChanged()` method, we can use the `getSelectedIndex()` method to get the item the user clicks and display the number of that item in the status bar:

```
   public void valueChanged(ListSelectionEvent e) {
   String msg = "You selected item no: ";
   msg += list.getSelectedIndex();
   showStatus(msg);
   }
```

When the user clicks an item in the list, the applet displays which item was clicked, as you can see in Figure 13.11:

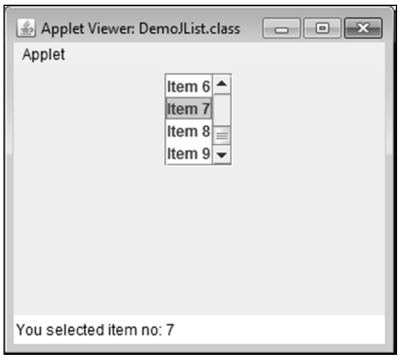


Figure 13.11: Handling Selections in a List Control

Note, that there's more than one way to select items in a list—you can make multiple selections in lists as well. See the next solution for the details.

Handling Multiple List Selections

The Product Support Specialist (PSS) appears and says, "Users are unhappy about your new program. They want to order a dozen products at once, but the list control in the program only lets them order one." "So?" you ask. "So the Big Boss is going to be very unhappy," the PSS says. "OK," you say, "I'll set up a multiple-selection list control."

List Selection Modes

By default, there are three selection modes for `JList` objects, and you can set the one you want to use with the `setSelectionMode()` method. The constants that you can pass to this method and what they mean are as follows:

- `SINGLE_SELECTION`—Single selection.
- `SINGLE_INTERVAL_SELECTION`—One interval selection. The user can select one and only one interval of items.
- `MULTIPLE_INTERVAL_SELECTION`—Multiple selection intervals.

We'll create a multiple-interval selection list control here and show how to determine what items are selected when the user makes a selection. To start, we just create a list control as in the previous solution, making it a multiple-interval selection list control. The code is as follows:

```
   import java.awt.*;
   import javax.swing.*;
   import javax.swing.event.*;
   /*<APPLET
      CODE=DemoMultipleSelection.class
      WIDTH=300
```

```
    HEIGHT=200 >
  </APPLET>*/
  public class DemoMultipleSelection extends JApplet implements ListSelectionListener {
          JList list;
          public void init() {
    Container c = getContentPane();
    String[] items = new String[10];
    for(int loop_index = 0; loop_index <= 9; loop_index++) {
    items[loop_index] = "Item " + loop_index; }
    list = new JList(items);
    JScrollPane scrollpane = new JScrollPane(list);
    list.setVisibleRowCount(5);
    list.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
    list.addListSelectionListener(this);
    c.setLayout(new FlowLayout());
    c.add(scrollpane); }
    /* some code */
  }
```

When the user makes a selection, we can determine what items were selected in the `valueChanged()` method by using the `getSelectedIndices()` method, which returns an array of the selected indexes. The end result is that we can report these indexes in the applet's status bar as follows:

```
  public void valueChanged(ListSelectionEvent e) {
  int[] indexes = list.getSelectedIndices();
  String msg = "You selected:";
  for(int loop_index = 0; loop_index < indexes.length; loop_index++) {
          msg += " item " + indexes[loop_index]; }
    showStatus(msg);
  }
```

You can see the result in Figure 13.12:

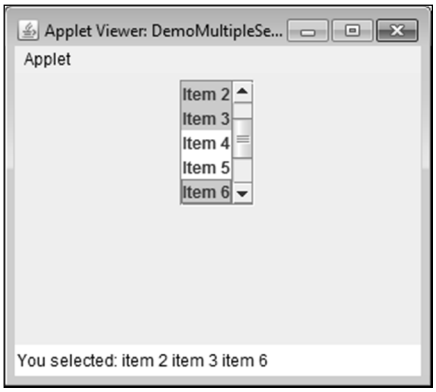


Figure 13.12: Handling Multiple Selections in a List Control

The user can click an item to select it and then use the Ctrl key to select other items or the Shift key to select an interval of items. Whenever a selection is made, the applet reports that selection in its status bar, as you see in Figure 13.12.

Displaying Images in Lists

"We have just got to jazz things up," the Big Boss says. "The competition is getting too tough." "Well," you ask, "how about adding more features?" "That costs money," the Big Boss growls. "Hmm," you say, "I suppose I could display images in the list controls." "Swell," says the Big Boss, "get to it."

Getting a Swing list control to display images is not as easy as adding images to a Swing label. In particular, you have to create your own model and renderer class for the list control, and we'll do that here. The result we are aiming for appears in Figure 13.13:

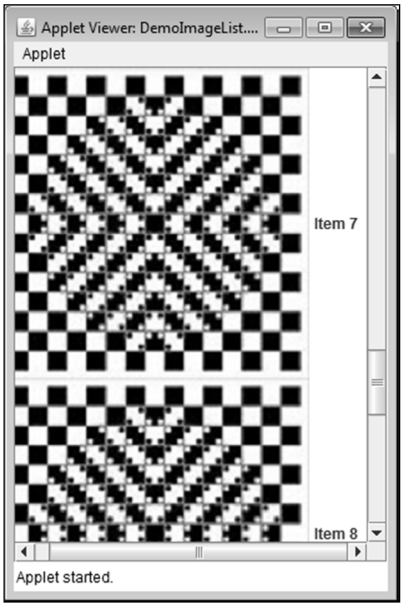


Figure 13.13: Adding Images to a List Control

To handle images in a list control, we create a new list control model, `MyModel`, and a new list control cell renderer, `MyRenderer`. To install the new model, which will hold the string for each list item and an image icon for each item, we can pass an object of the `MyModel` class to the `JList` constructor. To install the new renderer, we can use the `setCellRenderer()` method. The code is as follows:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
   CODE=DemoImageList.class
   WIDTH=300
   HEIGHT=400 >
</APPLET>*/
public class DemoImageList extends JApplet {
public void init() {
Container c = getContentPane();
MyModel model = new MyModel();
MyRenderer renderer = new MyRenderer();
JList list = new JList(model);
list.setCellRenderer(renderer);
list.setVisibleRowCount(5);
c.add(new JScrollPane(list)); }
}
```

This installs the new model and renderers—all that's left to do is to actually create the corresponding classes, and we'll do that in the next two solutions.

Creating a Custom List Model

Creating a new model for a list is not difficult; all you have to do is store your data by using the `addElement()` method. In this case, we'll create the new model for the list example in the previous solution, which will display images and text in a list control. In this case, we just make each data element in the model a two-element array; the first array element will hold the text for the list item, and the second array element will hold the image icon for the list item. The code is as follows:

```
class MyModel extends DefaultListModel {
   public MyModel() {
```

```
      for(int loop_index = 1; loop_index <= 10; loop_index++) {
      addElement(new Object[] {"Item " + loop_index,
      new ImageIcon("bulg.jpg")});
    }
      }
    }
```

That's all it takes. Now, a new model for a list control has been created. To actually draw the icon image, you have to handle the cell rendering yourself. We'll do just that in the next solution.

Creating a Custom List Cell Renderer

Each list item is actually of the class `JLabel`, so you can use the methods of `JLabel`, such as `setText()` and `setIcon()`, to display images in a list's cells. Here, we'll do that by creating a new cell renderer that will draw the items in the list we have developed over the previous two solutions. To create a cell renderer, you have to implement the `ListCellRenderer` interface, which has only one method: `getListCellRendererComponent()`. This object is passed to the object to render (which in this case is the two-element array holding the item's text and icon) and returns a cell renderer. The code is as follows (note that we also indicate whether an item is selected by adjusting its background):

```
    class MyRenderer extends JLabel implements ListCellRenderer {
    public MyRenderer() { setOpaque(true); }
    public Component getListCellRendererComponent(
    JList list, Object obj, int index, boolean isSelected,
    boolean focus) {
    MyModel model = (MyModel)list.getModel();
    setText((String)((Object[])obj)[0]);
    setIcon((Icon)((Object[])obj)[1]);
    if(!isSelected) {
    setBackground(list.getBackground());
    setForeground(list.getForeground()); }
    else {
    setBackground(list.getSelectionBackground());
    setForeground(list.getSelectionForeground()); }
    return this; }
    }
```

That's all it takes. Now, the list control will display images, as you see in Figure 13.13.

Handling Double Clicks in Lists

"Hey," says the NP, "I want to let the user launch an item from a list control by double-clicking it, but the `ListSelectionListener` interface's `valueChanged()` method is called for single clicks, and I can't tell whether the list has been double-clicked." "Well," you say, "you can handle mouse clicks yourself to intercept double clicks." The NP says, "Really? Tell me more!"

If you intercept mouse events, you can determine the number of mouse clicks, and by accessing the list's model, you can determine what list item was clicked.

Let's look at this in code. Here, we just add a list with a number of items in it to an applet and then add a mouse listener to the list:

```
    import java.awt.*;
    import javax.swing.*;
    import java.awt.event.*;
    /*<APPLET
      CODE=DemoListDoubleClick.class
      WIDTH=300
      HEIGHT=200 >
    </APPLET>*/
    public class DemoListDoubleClick extends JApplet implements MouseListener {
            public void init() {
    Container c = getContentPane();
    String[] items = {"Item 1", "Item 2", "Item 3", "Item 4",
```

```
    "Item 5", "Item 6", "Item 7", "Item 8",
    "Item 9", "Item 10"};
    JList list = new JList(items);
    c.setLayout(new FlowLayout());
    c.add(new JScrollPane(list));
    list.addMouseListener(this); }
    /* some code */
    }
```

Now, in the `mouseClicked()` method, we determine where the mouse was double clicked, then we use the `locationToIndex()` method of the list to determine what list item was clicked. Finally, we can determine how many times the item was clicked with the `MouseEvent's` method `getClickCount()` and therefore, report the number of times each item is clicked, as shown in the following code:

```
    public void mouseClicked(MouseEvent e) {
    JList list = (JList)e.getSource();
    int index = list.locationToIndex(e.getPoint());
    String msg = new String("Item " +
    ++index + " clicked " + e.getClickCount()+" times");
    showStatus(msg); }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
    public void mousePressed(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }
```

The result appears as shown in Figure 13.14:

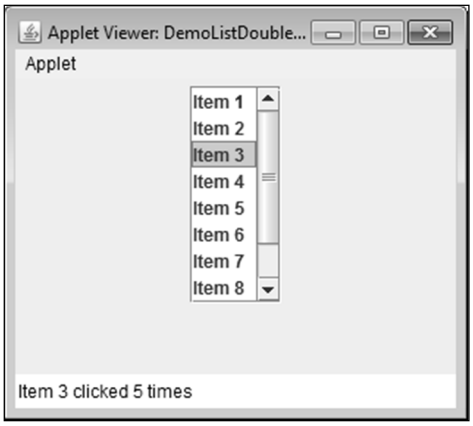


Figure 13.14: Handling Double Clicks in a List Control

You can see in the figure that the applet reports the number of times a list item is clicked. This way, you can handle double and even triple clicks.

Drag and Drop Support

Java provides a drag and drop support for creating a Window-based application. This support not only helps a programmer to give the drag and drop facility in the application but also helps to package the particular `JComponent` and move the same to another location. However, this support had undergone some changes since Java SE 6. There are two enhancements in this area, which are as follows:

- To avoid the use of selection for specifying drop location, drop modes are customized
- At the time of transferable operations, additional information can be obtained

The drag and drop features are used by the `JList`, `JTable`, `JTextComponent`, and `JTree` components. They have a new `setDropMode()` method, which accepts a `DropMode` enum constant described in Table 13.16; each component has a specific set of drop modes considered as acceptable:

Table 13.16: Enum Constant Summary	
Constant	**Does this**
INSERT	It tracks the drop location in terms of the position, where new data is inserted
INSERT_COLS	It tracks the drop location in terms of the column index, where new column data is inserted to accommodate the dropped data
INSERT_ROWS	It tracks the drop location in terms of the row index, where new row data is inserted to accommodate the dropped data
ON	It tracks the drop location in terms of the index of existing items
ON_OR_INSERT	It specifies that the data can be dropped on the existing items or in the insert locations as specified by INSERT
ON_OR_INSERT_COLS	It specifies that the data can be dropped on the existing items or in the insert columns as specified by INSERT_COLS
ON_OR_INSERT_ROWS	It specifies that the data can be dropped on the existing items or in insert rows as specified by INSERT_ROWS
USE_SELECTION	It tracks the drop location with the help of the component's own internal selection mechanism

All the components support a drop mode of USE_SELECTION. This is the way of indicating where to drop something. A DropMode of ON is supported by the JList, JTable, and JTree components. The JTable component has the following four additional drop-mode options:

- INSERT_COLS
- INSERT_ ROWS
- ON_OR_INSERT_COLS
- ON_OR_INSERT_ROWS

The inheritance diagram for the DropMode class is as follows:

```
java.lang.Object
  |_ java.lang.Enum<DropMode>
        |_  javax.swing.DropMode
```

While dragging and dropping, a drop mode determines the way a component tracks and shows the drop location. You can find the methods of the DropMode class in Table 13.17:

Table 13.17: Methods of the DropMode Class	
Method	**Does this**
static DropMode valueOf(String name)	It returns the enum constant of this type with the specified name
static DropMode[] values()	It returns an array containing the constants of this enum type in the order they are declared

Various drop-mode options are signified by the DemoDragDrop class. In the JTextField, you are allowed to enter and highlight anything, and then drag it over the JTree. After that, you can drop the text and view the different behaviors for various settings of the mode.

The DemoDragDrop class describes different drop-mode options. Consider the following code:

```
import javax.swing.*;
import javax.swing.tree.*;
import java.awt.*;
import java.awt.datatransfer.*;
import java.awt.event.*;
import java.io.*;
public class DemoDragDrop {
public static void main(String args[]) {
   Runnable run = new Runnable() {
```

```
        public void run() {
                JFrame frm = new JFrame(" Demo Drag And Drop");
                frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                JPanel topPanel = new JPanel(new BorderLayout());
                JLabel dragLabel = new JLabel("Drag Me:");
                JTextField text = new JTextField();
                text.setDragEnabled(true);
                topPanel.add(dragLabel,BorderLayout.WEST);
                topPanel.add(text,BorderLayout.CENTER);
                frm.add(topPanel,BorderLayout.NORTH);
                final JTree tree = new JTree();
                final DefaultTreeModel model =
                (DefaultTreeModel)tree.getModel();
                tree.setTransferHandler(new TransferHandler () {
                public boolean canImport(TransferHandler.TransferSupport
                support) {
                 if(!support.isDataFlavorSupported(DataFlavor.stringFlavor) ||
                !support.isDrop()) {
                                return false; }
                        JTree.DropLocation location =
                        (JTree.DropLocation)support.getDropLocation();
                        return location.getPath() != null; }
                public boolean importData(TransferHandler.
                TransferSupport support) {
                if(!canImport(support)) { return false; }
                        JTree.DropLocation location =
                        (JTree.DropLocation)support.getDropLocation();
                        TreePath treePath = location.getPath();
                        Transferable transfer = support.getTransferable();
                        String transferData;
                                try {
                                transferData=
                                (String)transfer.getTransferData
                                (DataFlavor.stringFlavor); }
                        catch(IOException e) { return false; }
                        catch(UnsupportedFlavorException e) {
                                return false; }
                        int childIndex = location.getChildIndex();
                        if(childIndex == -1) {
                                childIndex =
                                model.getChildCount
                                (treePath.getLastPathComponent()); }
                        DefaultMutableTreeNode newNode =
                        new DefaultMutableTreeNode(transferData);
                        DefaultMutableTreeNode parentNode=
                        (DefaultMutableTreeNode)treePath.getLastPathComponent();
                        model.insertNodeInto(newNode,parentNode,childIndex);
                        TreePath newTreePath = treePath.pathByAddingChild(newNode);
                        tree.makeVisible(newTreePath);
                        tree.scrollRectToVisible
                        (tree.getPathBounds(newTreePath));
                        return true; }});
                JScrollPane scrollpane = new JScrollPane(tree);
                frm.add(scrollpane, BorderLayout.CENTER);
                JPanel bottomPanel = new JPanel();
                JLabel comboLabel = new JLabel("DropMode");
                String options[]
                = {"USE_SELETION","ON","INSERT","ON_OR_INSERT"};
                final DropMode mode[] =
                        { DropMode.USE_SELECTION,DropMode.ON,DropMode.
                INSERT,DropMode.ON_OR_INSERT};
                final JComboBox comboBox = new JComboBox(options);
                comboBox.addActionListener(new ActionListener() {
                        public void actionPerformed(ActionEvent e) {
                                int selectedIndex =
```

```
                                    comboBox.getSelectedIndex();
                                    tree.setDropMode(mode[selectedIndex]);
                                }});
                        bottomPanel.add(comboLabel);
                        bottomPanel.add(comboBox);
                        frm.add(bottomPanel,BorderLayout.SOUTH);
                        frm.setSize(300,400);
                        frm.setVisible(true); }
        };
        EventQueue.invokeLater(run); }
    }
```

The output of this code shows the program window and dragging and dropping of the items of the JTree, as shown in Figure 13.15(a) and Figure 13.15(b), respectively:

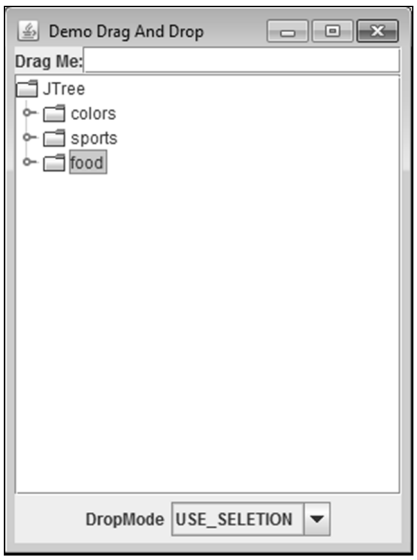


Figure 13.15(a): Displaying the JTree

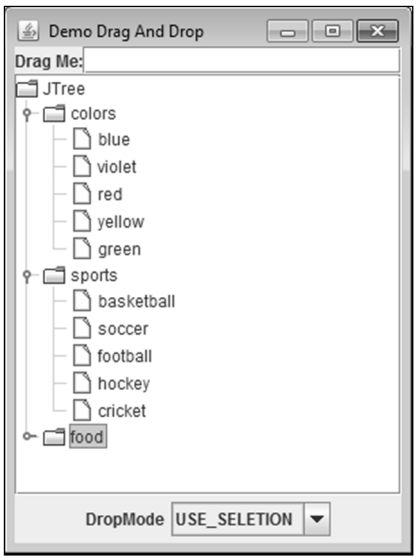


Figure 13.15(b): Showing the Drag and Drop Tree

It is aforementioned that you can type something in the text field of the program window, highlight it, and drag it over the `JTree`.

Creating a Table

The Big Boss is concerned and wants to give flexibility to the programmer. Big Boss says, "The table in Swing is important. Is there any complexity in handling the data?" "No problem Boss" you say. "JDBC will help here to fight with complexities. It gives much more flexibility to the programmer. It's easy to create a simple `JTable`."

A `JTable` displays similar or dissimilar type of data in the form of a rectangular array. The data contained in each column can be of basic or class type. Although a `JTable` can be created in a number of ways, the most general and convenient way is to create an object that encapsulates the data that the `JTable` would display. A `JTable` can efficiently display the data stored in a database and has to implement the `javax.swing.TableModel` interface for the same. The interfaces dealing with `javax.swing.JTable` are listed in Table 13.18:

Table 13.18: Interfaces of JTable

Interface	Does this
`TableCellEditor`	This interface defines the method that any object needs to implement. Here, the object would like to be an editor of values for components, such as `JListBox`, `JComboBox`, `JTree`, or `JTable`

Table 13.18: Interfaces of JTable	
Interface	**Does this**
`TableCellRenderer`	This interface defines the method required by any object that would like to be a renderer for cells in a `JTable`
`TableColumnModel`	It defines the requirements for a table column model object suitable for use with `JTable`
`TableModel`	The `TableModel` interface specifies the methods that the `JTable` will use to interrogate a tabular data model

To create an object of `JTable`, you just need to pass the `TableModel` reference in the constructor. This is shown as follows:

```
JTable table = new JTable(model);
```

Here, the variable `model` holds the object reference that encapsulates the data to be displayed in the `JTable`. Next is the implementation of the `TableModel` interface that lies along with the `JTable` class in the `javax.swing.table` package and contains a total of nine methods. An abstract class, `AbstractTableModel`, implements six of the nine methods. Therefore, if you extend the `AbstractTableModel` class, three methods still need to be implemented.

The class that implements the `TableModel` interface actually encapsulates the `ResultSet` object. You'll study about `ResultSet` in Chapter 24.

The full set of methods declared in the `TableModel` interface is presented in Table 13.19:

Table 13.19: Methods of the TableModel class	
Method	**Does this**
`void addTableModelListener (TableModelListener l)`	It adds a listener to the list that is notified each time a change to the data model occurs
`Class<?> getColumnClass(int columnIndex)`	It returns the most specific superclass for all the cell values in the column
`int getColumnCount()`	It returns the number of columns in the model
`String getColumnName(int columnIndex)`	It returns the name of the column at columnIndex
`int getRowCount()`	It returns the number of rows in the model
`Object getValueAt(int rowIndex, int columnIndex)`	It returns the value for the cell at columnIndex and rowIndex
`boolean isCellEditable(int rowIndex, int columnIndex)`	It returns True if the cell at rowIndex and columnIndex is editable
`void removeTableModelListener (TableModelListener l)`	It removes a listener from the list that is notified each time a change to the data model occurs
`void setValueAt(Object aValue, int rowIndex, int columnIndex)`	It sets the value in the cell at columnIndex and rowIndex to aValue

Note that it is the `JTable` object that actually calls these methods to access and manipulate the data in the table model. While creating a class that implements the `TableModel` interface, it's better to make it capable of replacing the `ResultSet` object at any point of time. This will make a single `JTable` object capable of displaying data corresponding to different resultsets just by assigning the reference of another `ResultSet` object to the `TableModel` object. It's easy to accomplish this functionality; you just have to define a method that accepts a `ResultSet` object as argument and make the data available with the help of the `TableModel` interface, as shown in the following code:

```
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import javax.swing.table.AbstractTableModel;
import java.util.Vector;
class DemoTableModel extends AbstractTableModel {
```

```
      public void setResultSet (ResultSet rs) { }
            public int getColumnCount() { }
            public int getRowCount() { }
            public String getValueAt (int row, int column) { }
            public String getColumnName (int column) { }
   }
```

You can return to the actual `ResultSet` object for accessing the data that is returned by the `TableModel` interface. However, this will result in making a trip to the database each time. Therefore, it's better to avoid the database trips by caching the data from the `ResultSet` object in a `DemoTableModel` object. To ensure this, the `setResultSet()` method will have to set it in order. Two sets of information are stored in the `DemoTableModel` object. One is the names (`String` objects) and second is the data (can be of any type, matching the Structured Query Language (SQL) data types) in the table. For simplicity, the `getString()` method is used to access the data. You can extract any SQL data type as a `String`, and at this point, you'll only focus upon strings.

Table Sorting and Filtering

Java SE made sorting and filtering of the table data much easier. A *sorter* object manages table sorting and filtering. With the help of different classes and interfaces, users can simply click on a column to sort elements. In addition, filtering lets you display only those table rows that match with the user-supplied criteria.

Sorting Rows

For sorting and filtering rows, we use an abstract `RowSorter` class for supporting two mappings. One is for mapping rows in a `JTable` to the elements of the underlying model and the other is for mapping from the underlying model to the rows. It works with both `TableModel` and `ListModel`.

The Java SE provides a `TableRowSorter` class to work with `JTable`. The reference of class object implementing the `TableModel` interface is passed to the `TableRowSorter` constructor and then the created `RowSorter` is passed to the `setRowSorter()` method of `JTable`.

An example for the sorting table class `DemoTableSort` is as follows:

```
import javax.swing.*;
import javax.swing.table.*;
import java.awt.*;
public class DemoTableSort {
   public static void main(String args[]) {
         Runnable run = new Runnable() {
               public void run() {
                     JFrame frm = new JFrame("Sorting Table");
               frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                     Object rows[][] = {
                           {"AMZN", "Amazon", 43.28},
                           {"GOOG", "Google", 389.33},
                           {"MCRSFT", "Microsoft", 28.66},
                           {"SANSFT", "Sansoft Inc.", 128.76},
                           {"NOK", "Nokia Corp", 19.73},
                           {"ORCL", "Oracle Corp.", 14.58},
                           {"SUNMCRS", "Sun Microsystems Inc.", 5.97},
                           {"VODA",  "Vodafone Group", 39.92},
                           {"YHO", "Yahoo!", 38.67}
                     };
                     String columns[] = {"Symbol", "Name", "Price"};
                     TableModel model =
                     new DefaultTableModel(rows, columns) {
                     public Class getColumnClass(int column) {
                           Class returnValue;
                           if ((column >= 0) &&
                           (column < getColumnCount())) {
```

```
                                        returnValue = getValueAt(0,
                                        column).getClass(); }
                                else { returnValue = Object.class; }
                                return returnValue; }
                    };
                    JTable table = new JTable(model);
                    RowSorter<TableModel> rsorter =
                    new TableRowSorter<TableModel>(model);
                    table.setRowSorter(rsorter);
                    JScrollPane scrollpane = new JScrollPane(table);
                    frm.add(scrollpane, BorderLayout.CENTER);
                    frm.setSize(300, 150);
                    frm.setVisible(true); }
            };
            EventQueue.invokeLater(run); }
    }
```

When you click on one of the column of the displayed table, you'll get the contents of that column reordered. In the following code, we are describing some important lines that are pertinent to sorting:

```
JTable table = new JTable(model);
RowSorter<TableModel> rsorter =
new TableRowSorter<TableModel>(model);
table.setRowSorter(rsorter);
```

The first line is used to associate the model with the table. The second line is then used for creating a `RowSorter` according to the model. The third line is used for associating the `RowSorter` with the `JTable`. When a user clicks on the column header, it sorts that column. If the same column header is clicked for a second time, it reverses the sorted order.

Here, you'll get three figures that describe the sorting of the table data. Figure 13.16 shows the table of three columns:

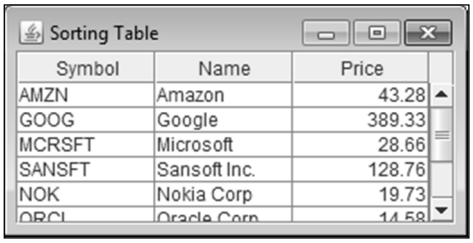


Figure 13.16: Sorting Table when the Column Header Has Not Been Clicked

It is the initial table when the column header has not been clicked.

Figure 13.17 shows the table when a column header "Name" has been clicked once, and the names are sorted in an alphabetical order:

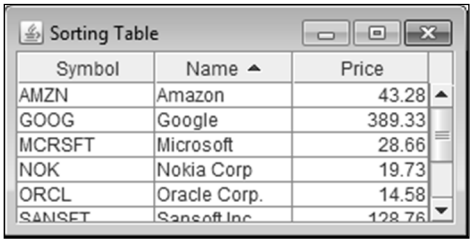


Figure 13.17: Sorting Table when the Column Header "Name" Has Been Clicked Once

Figure 13.18 shows the table when the column header "Name" has been double clicked; it sorts the names in reverse of alphabetic order:

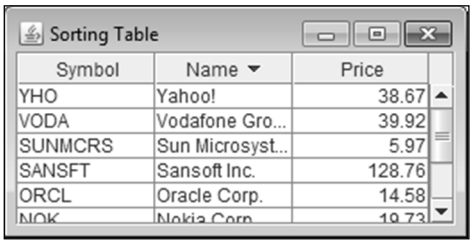


Figure 13.18: Sorting Table when the Column Header "Name" Has Been Double Clicked

Filtering Table Rows

Filtering is used for displaying rows in a table that resembles with the criteria supplied by the user. An abstract `RowFilter` class is used to filter the table contents. It is associated with the `TableRowSorter` class and contains only one method for filtering, that is:

```
boolean include(RowFilter.Entry<? extends M,? extends I> entry)
```

For every entry in the model related with the `RowSorter`, the specified entry should be displayed in the present view of the model specified by the `include()` method. In various cases, it is not mandatory to build your own `RowFilter` implementation. The following six static methods are used by the `RowFilter` class for creating filters:

```
andFilter(Iterable<? extends RowFilter<? super M,? super I>> filters)
dateFilter(RowFilter.ComparisonType type, Date date, int... indices)
notFilter(RowFilter<M,I> filter)
numberFilter(RowFilter.ComparisonType type, Number number, int... indices)
orFilter(Iterable<? extends RowFilter<? super M,? super I>> filters)
regexFilter(String regex, int... indices)
```

For the `RowFilter` factory methods that have an argument of indices (`dateFilter`, `numberFilter`, `regexFilter`), only the set of columns corresponding to the specified indices are checked in the model. If it does not specify any indices, all columns are checked for a match by using only of the following methods:

- `dateFilter`—It checks for a matching date
- `numberFilter`—It checks for a matching number
- `notFilter`—It is used for reversing another file
- `andFilter` and `orFilter`—It is used for logically combining other filters
- `regexFilter`—It uses a regular expression for filtering

The `DemoTableFilter` class in the following code uses a `regexFilter` to filter table content:

```
import javax.swing.*;
import javax.swing.table.*;
import java.awt.*;
import java.awt.event.*;
import java.util.regex.*;
public class DemoTableFilter {
  public static void main(String args[]) {
        Runnable run = new Runnable() {
                public void run() {
                        JFrame frm = new JFrame("Sorting Table");
                        frm.setDefaultCloseOperation
                        (JFrame.EXIT_ON_CLOSE);
                        Object rows[][] = {
                                {"AMZN", "Amazon", 43.28},
                                {"GOOG", "Google", 389.33},
                                {"MCRSFT", "Microsoft", 28.66},
                                {"SANSFT", "Sansoft Inc.", 128.76},
```

```
                                        {"NOK", "Nokia Corp", 19.73},
                                        {"ORCL", "Oracle Corp.", 14.58},
                                        {"SUNMCRS", "Sun Microsystems Inc.", 5.97},
                                        {"VODA",  "Vodafone Group", 39.92},
                                        {"YHO", "Yahoo!", 38.67}
                                };
                                Object columns[] = {"Symbol", "Name", "Price"};
                                TableModel model =
                                new DefaultTableModel(rows, columns) {
                                        public Class getColumnClass(int column) {
                                                Class returnValue;
                                                if ((column >= 0) &&
                                                (column < getColumnCount())) {
                                                        returnValue =
                                                        getValueAt(0,
                                                        column).getClass();
                                                }else { returnValue = Object.class; }
                                                return returnValue; }
                                };
                                JTable table = new JTable(model);
                                final TableRowSorter<TableModel> rsorter =
                                new TableRowSorter<TableModel>(model);
                                table.setRowSorter(rsorter);
                                JScrollPane scrollpane = new JScrollPane(table);
                                frm.add(scrollpane, BorderLayout.CENTER);
                                JPanel jp = new JPanel(new BorderLayout());
                                JLabel label = new JLabel("Filter");
                                jp.add(label, BorderLayout.WEST);
                                final JTextField filterText =
                                new JTextField("MCR");
                                jp.add(filterText, BorderLayout.CENTER);
                                frm.add(jp, BorderLayout.NORTH);
                                JButton button = new JButton("Filter");
                                button.addActionListener(new ActionListener() {
                                public void actionPerformed(ActionEvent e) {
                                                String text = filterText.getText();
                                                if (text.length() == 0) {
                                                        rsorter.setRowFilter(null);
                                                } else {
                                                        try {
                                                                rsorter.setRowFilter(
                                                        RowFilter.regexFilter(text));
                                                        }
                                                        catch
                                                        (PatternSyntaxException
                                                        pse) {
                                                                System.err.println
                                                                ("Bad regex pattern");
                                                        }
                                                }
                                        }
                                });
                                frm.add(button, BorderLayout.SOUTH);
                                frm.setSize(400, 300);
                                frm.setVisible(true); }
                        };
                        EventQueue.invokeLater(run); }
        }
```

Here, we have specified the string "MCR" for filtering. The display sets a filter for all strings with the characters `MCR` somewhere in them. The characters "`^`" and "`$`" are used for testing exact matches at the beginning and end of the string, respectively. Whenever the user presses the "`Filter`" button at the bottom, the `Matcher.find()`

method is used internally for inclusion testing. The results of this code before and after filtering are displayed in Figures 13.19 and 13.20, respectively:



Figure 13.19: Sorting Table before Filtering

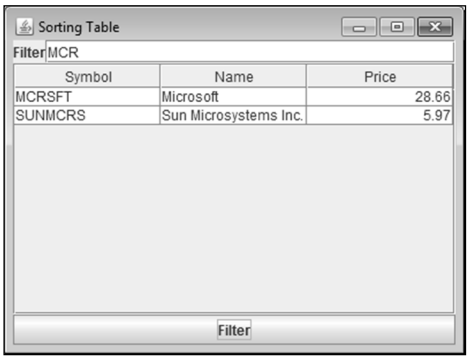


Figure 13.20: Sorting Table after Filtering

Adding Rows and Columns to Tables at Runtime

The Programming Correctness Czar appears and says, "OK, with the help of Java we can create Tables." "Well, they don't seem to be very professional—with professional controls, you could add rows and columns to tables." "No problem," you say. "It is possible in Java. Even here, a vector object will provide sufficient flexibility to accommodate as per the requirement." "Oh," says the Programming Correctness Czar.

After creating a table by the `TableModel` interface, you can add columns and rows to the table. Defining `column_heading` as an array with zero elements ensures that you start out with a non-null array even though there is no `ResultSet` initially. To store a column name in a `String` array, you can add a data member to the `DemoTableModel` class as:

```
private String[] column_heading = new String[0];
```

It is quite impossible to know the number of rows and columns of data in advance. Therefore, you won't want to use an array. A `Vector<>` object will provide sufficient flexibility to accommodate as per the requirement. You can store the content of a row as an array of `String` objects that you can store as an element in the `Vector<>` object. (You'll learn more about the `Vector` class in Chapter 21.) The data member that stores the rows of data value is defined as follows:

```
private Vector<String[]> Rows_data = new Vector<String[]>();
```

It is quite imperative to set the type for all the data items in a row to `String`. However, if you want to accommodate different types within a row, you could use an array of type `Object[]` to store each row. Here, you need to remember the `import` statement for the `Vector` class, which is defined in the `java.util` package. Now, you can implement the `setResultSet()` method in the `DemoTableModel` class as follows:

```
public void setResultSet (ResultSet results) {
        try {
        ResultSetMetaData metadata = results.getMetaData();
        int columns = metadata.getColumnCount();
        column_heading = new String[columns];
        for (int i = 0; i<columns i++) {
                Column_heading[i] = metadata.getColumnLabel (i+1); }
        Rows_data.clear();
        String[] row_data;
        while (results.next()) {
                row_data = new String[columns];
                for (int i = 0; i < columns; i++) {
                        row_data[i] = results.getString (i+1); }
  Rows_data.addElement (row_data); }
  fireTableChanged (null); }
  catch (SQLException e) { System.err.printin(e); }
}
```

Acquiring access to the `ResultSetMetaData` (will learn more about `ResultSetMetaData` in Chapter 24) interface object corresponding to the `ResultSet` object is important for retrieving the names and number of columns. You can use the `getColumnLabel()` method to obtain a label to be used for the column name. The label depicts either the column name known to the database or the name you supply in the `SELECT` statement to create the `ResultSet` object. The names of the columns are stored in the `column_heading`.

To remove any existing object references in the `Vector<String[]>`, you can use the `clear()` method. Each vector element is a reference to an array of `String` objects, `row_data`, which is created inside the `while` loop. The array elements are assigned values inside the `for` loop.

Once the array gets created, you store a reference to it in `rows_data`. After all the rows from the `ResultSet` object have been stored, you call the `fireTableChanged()` method that the `DemoTableModel` class inherits from the base class. This method notifies all listeners for the `JTable` object for this model that the model has changed. Therefore, the `JTable` object should redraw itself from scratch. The argument of the `fireTableChanged()` method is a reference to an object of the `TableModelEvent` type that can be used for recording the portions of the model that have been modified.

Creating Trees

The NP is unhappy and says, "Java might comprise very limited options for arranging data. What will be the simple method to display required data as simple as it can be?'' "No Sir", you say. "It is possible with the `JTree` class. The `JTree` class uses a `TreeModel` to get its data. It displays data organized in a tree-like structure."

The interfaces for dealing with `javax.swing.JTree` are enlisted in Table 13.20:

Table 13.20: Interfaces of the JTree class

Interface	Does this
`ImageObserver`	It observes the constructed images and gets the notifications about image information
`MenuContainer`	It is the super class of all menu-related containers
`Serializable`	It enables serializability of class by implementing the java.io.Serializable interface
`Accessible`	It is the main interface for the accessibility package
`Scrollable`	It provides information to scrolling containers, e.g., JScrollPane

An object of a class that implements the `TreeNode` interface is called a node. The methods declared by the `TreeNode` interface that provide the mean of navigating a tree are enlisted in Table 13.21:

Table 13.21: Methods of the TreeNode interface

Method	Does this
`TreeNode getChildAt(int childIndex)`	It returns the child TreeNode at index childIndex
`int getChildCount()`	It returns the number of children tree nodes the receiver contains
`boolean isLeaf()`	It returns True if the receiver is a leaf
`Enumeration children()`	It returns the children of the receiver as an Enumeration
`boolean getAllowsChildren()`	It returns True if the receiver allows children
`int getIndex(TreeNode node)`	It returns the index of node in the receiver's children
`TreeNode getParent()`	It returns the parent TreeNode of the receiver

The `TreeNode` interface does not provide enough ability for linking nodes in a tree. Each node must refer to its parent node so that the tree structure gets created. The `MutableTreeNode` interface extends the `TreeNode` interface and provides declaration of methods to construct the tree. The different methods of the `MutableTreeNode` are listed in Table 13.22:

Table 13.22: Methods of MutableTreeNode	
Method	**Does this**
`void insert(MutableTreeNode child, int index)`	It adds child to the receiver at index
`void remove(int index)`	It removes the child at index from the receiver
`void remove(MutableTreeNode node)`	It removes node from the receiver
`void removeFromParent()`	It removes the receiver from its parent
`void setParent(MutableTreeNode newParent)`	It sets the parent of the receiver to newParent
`void setUserObject(Object object)`	It resets the user object of the receiver to object

As per your requirement, you can create your own class to define nodes. In most of the cases, the `DefaultMutableTreeNode` class in the `javax.swing.tree` package is sufficient. This class implements the `MutableTreeNode` interface and also provides a few methods of its own. The default constructor of this class creates a node with no parent and children.

The `DefaultMutableTreeNode` class's objects are most commonly used to create nodes of a tree. A "user object" (a value supplied by the user in the `DefaultMutableTreeNode` constructor) acts as the value displayed at each node. The `toString()` method of the "user object" actually represents what is displayed for each node.

After you have created the nodes, organize them in a tree structure by using the `parentNode.add(childNode)` method. The node is finally passed in the `JTree` constructor. You usually add a `JScrollPane` element to support the change in size of the tree due to node expansion and collapse. The following code represents a simple tree structure:

```
DefaultMutableTreeNode root = new DefaultMutableTreeNode("Rootnode");
DefaultMutableTreeNode child1 = new DefaultMutableTreeNode("Child 1");
root.add(child1);
DefaultMutableTreeNode child2 = new DefaultMutableTreeNode("Child 2");
root.add(child2);
JTree tree = new JTree(root);
someWindow.add(new JScrollPane(tree));
```

When you create more complicated trees, it's tough to maintain then when everything is done "by hand." Therefore, it's better you first create a simple tree-like data structure and then use that data structure to create and add other nodes.

Adding Data to Trees

The NP appears and says, "I need to know how to add data to trees.'' "Ok'', you say. "This is very simple. The easiest approach is to start with the `DefaultMutableTreeNode` class. Let me tell you more."

A `JTree` gets its data by using the `TreeModel` interface. Like a `JList`, you can have a custom model to replace the existing tree model and specify the way to extract data. A default `TreeModel` uses the `TreeNode` interface to hold the tree data. You should prefer creating a custom `TreeNode` interface and leave the `TreeModel` interface unchanged. Beginning with the `DefaultMutableTreeNode` class is considered as the most convenient way.

For example, the following code represents a potentially infinite tree in which a section in an outline is described by each node. The root node is "1," the first-level children are 1.1, 1.2, 1.3, etc., and the second-level children are 1.1.1, 1.1.2, etc.. The command-line argument supplied determines the actual number of child node each nodes that possesses.

Note that the `getChildCount()` method of the `DefaultMutableTreeNode` class is called before any of the child nodes are retrieved. Next, to indicate whether a child node has been created or not, a flag is kept. After the `getChildCount()` method is called and if the flag is found to be false, the child nodes are created and added

to the tree. You override the `isLeaf()` method so that the tree doesn't count and further create the nodes. The code of the `DemoJTree.java` file is as follows:

```
import java.awt.*;
import javax.swing.*;
public class DemoJTree extends JFrame {
  int n = 5;
  public static void main(String[] args) {
                int i = 5;
                if (args.length > 0)
        try { i = Integer.parseInt(args[0]); }
  catch(NumberFormatException e) {
  System.out.println("Can't parse number using defaultof "); }
        new DemoJTree(i); }
  public DemoJTree(int i) {
  super("Creating a Dynamic JTree");
        Container c = getContentPane();
        JTree tree = new JTree(new App(1, n));
        c.add(new JScrollPane(tree), BorderLayout.CENTER);
        setSize(300, 475);
        setVisible(true); }
}
```

The `DemoJTree.java` will execute only after creating the `App.java` file because the method of the App.java is being invoked. The result of the code is displayed in Figure 13.21:

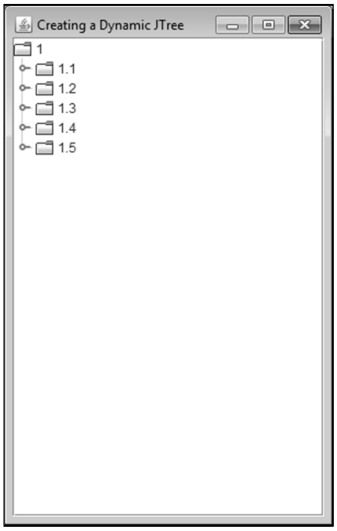


Figure 13.21: Creating a Dynamic Tree

The code for the `App.java` is as follows:

```
import java.awt.*;
import javax.swing.*;
import javax.swing.tree.*;
class App extends DefaultMutableTreeNode {
  private boolean ChildrenDefined = false;
  private int outline_num;
  private int nChildren;
  public App(int outline_num, int nChildren) {
        this.outline_num = outline_num;
        this.nChildren = nChildren; }
  public boolean isLeaf() { return(false); }
  public int getChildCount() {
        if (!ChildrenDefined)
        defineChild();
```

```
            return(super.getChildCount()); }
      private void defineChild() {
            ChildrenDefined = true;
            for(int i=0; i<nChildren; i++)
            add(new App(i+1, nChildren)); }
      public String toString() {
            TreeNode parent = getParent();
            if (parent == null)
            return(String.valueOf(outline_num));
            else
            return(parent.toString() + "." + outline_num);
   }
   }
```

The result of the code, `DemoJTree.java`, after expanding a few nodes is shown in Figure 13.22:

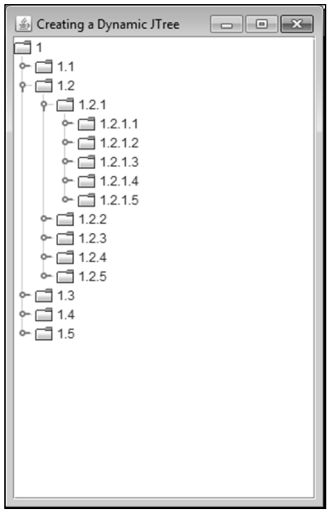


Figure 13.22: Showing Child Nodes of Tree

Handling Tree Events

A `TreeSelectionListener` interface is used to handle tree events and contains a single method, `valueChanged()`. You start with extracting the currently selected node by calling the `tree.getLastSelectedPathComponent()` method, followed by casting it to your node type, which is generally `DefaultMutableTreeNode`, and finally extracting the value (user object) at the node by calling the `getUserObject()` method. To retrieve the node label, you can just call the `toString()` method on the result of the `tree.getLastSelectedPathComponent()` method. An example is as follows:

```
   import java.awt.*;
   import javax.swing.*;
   import javax.swing.tree.*;
   import javax.swing.event.*;
   public class DemoTreeEvent extends JFrame
   implements TreeSelectionListener {
     public static void main(String[] args) { new DemoTreeEvent(); }
     private JTree tree;
     private JTextField currentSelectionField;
     public DemoTreeEvent() {
           super("JTree Selections");

           Container c = getContentPane();
           DefaultMutableTreeNode root =
           new DefaultMutableTreeNode("Root");
```

```
            DefaultMutableTreeNode childnode;
            DefaultMutableTreeNode grandChildnode;
            for(int childIndex=1; childIndex<4; childIndex++) {
            childnode = new DefaultMutableTreeNode("Child " + childIndex);
                    root.add(childnode);
                    for(int grandChildIndex=1; grandChildIndex<4;
                    grandChildIndex++) {
                            grandChildnode =
                            new DefaultMutableTreeNode("Grandchild " + childIndex + "."
                            + grandChildIndex);
                            childnode.add(grandChildnode); }
            }
            tree = new JTree(root);
            tree.addTreeSelectionListener(this);
            c.add(new JScrollPane(tree), BorderLayout.CENTER);
            currentSelectionField = new JTextField("Current Selection: NONE");
            c.add(currentSelectionField, BorderLayout.SOUTH);
            setSize(250, 275);
            setVisible(true); }
            public void valueChanged(TreeSelectionEvent event) {
                    currentSelectionField.setText ("Current Selection: " +
                    tree.getLastSelectedPathComponent().toString()); }
    }
```

The result of the code is shown in Figure 13.23:

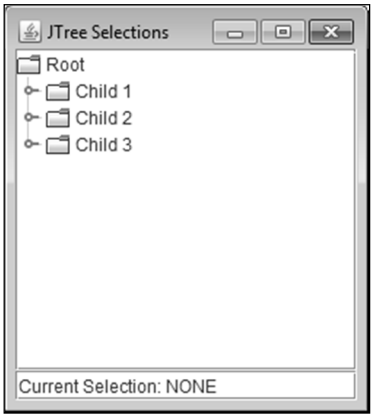


Figure 13.23: Showing Selection in Tree

The result of the code after expanding the tree and selecting a node is shown in Figure 13.24:

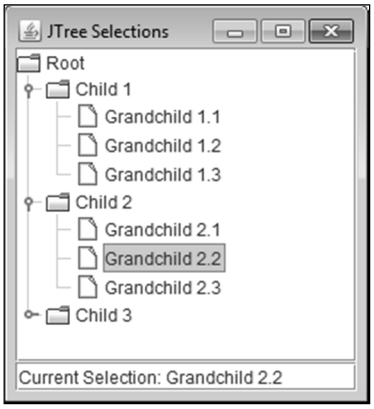


Figure 13.24: Showing Selection in Tree Branches

Summary

This chapter has provided a brief introduction of viewports, scrollpanes, sliders, scrollbars, lists, tables, and trees. Then, it has explained how to handle viewports, create scrollpanes, and scroll images. You have also learned how to create and fill a slider and to paint the slider labels and slider tick marks. The chapter has discussed how to create scrollbars, lists, and handle and display images in lists. Further, you have also learned to create a custom list model and custom list cell renderer. The chapter also explained how to handle double clicks in lists, implement drag and drop, and create a table.

In the next chapter, you will learn about Combo Boxes, Progress Bars, Tooltips, Separators, and Choosers.



14

Swing—Combo Boxes, Progress Bars, Tooltips, Separators, and Choosers

If you need an immediate solution to:	*See page:*
Creating Combo Boxes	492
Handling Combo Box Selection Events	496
Creating Editable Combo Boxes	497
Adding Images to Combo Boxes	499
Creating a Combo Box Model	500
Creating a Combo Box Custom Renderer	500
Creating Progress Bars	500
Updating Progress Bars	503
Handling Progress Bar Events	504
Creating Separators	507
Resizing Separators Automatically	508
Creating Color Choosers	510
Creating File Choosers	512
Creating File Chooser Filters	518

In Depth

In this chapter, we are going to cover some important Swing controls—combo boxes, progress bars, separators, and choosers. We'll also take a look at adding tooltips, those small windows that display explanatory text when you let the mouse rest over a control. All these topics, especially combo boxes, are important topics in Java, and we'll take a quick look at them before digging into the code.

Combo Boxes

The combo box is one of the most common controls in Graphical User Interface (GUI) programming, but the AWT doesn't have a combo box. Swing rectifies that. Combo boxes are combinations of text fields and drop-down lists. They are very useful in that they let the user either select an item from the list or enter own value into the text field. Combo boxes are also very compact controls—they only display one item and a button that the user can use to display the drop-down list of other items. Combo boxes can function as drop-down lists, presenting the user with an even more compact way of displaying a list of items than a list control could manage, and they have become very popular for that reason. In fact, Swing makes the default combo box into a drop-down list only because the default setting for combo boxes is to make them non-editable, which turns them into drop-down lists.

Progress Bars

Progress bars are relatively new controls, got popular as a way to give the user some indication of the progress of a long operation. Originally introduced as a way to show how an installation program is progressing, progress bars are now used for all kinds of time-consuming operations, such as downloading from the Internet. They display a colored bar inside them that grows (or shrinks), much like the mercury in a thermometer, to display visually how an operation is proceeding. You can orient progress bars horizontally or vertically, select the colors and labels used in them, and handle their events. Even so, progress bars remain simple controls, because they really only have one function—displaying the progress of some task. You'll see all the possibilities in this chapter.

Choosers

Like the AWT, Swing supports dialog boxes. Unlike the AWT, Swing also supports two dialog boxes that you don't have to create or customize yourself—file choosers and color choosers. File choosers let the user select a file for opening or saving, much like any standard file dialog box. Color choosers let the user select a color among many. Both these choosers represent standard dialog boxes, and Sun has simply saved your time by creating them for you. We'll work with both these choosers in this chapter. We can use the color chooser in programs immediately; but to actually use the file names returned by file choosers, we will have to wait until we start working with files in a few chapters.

Tooltips

Tooltips are the small windows that appear and display explanatory text (such as Download Now or Open New Folder) when the mouse rests over a control. Tooltips can be very useful because GUI users have a great resistance to reading manuals. All the user has to do is let the mouse ride over your program to see what the various controls do. On the other hand, bear in mind that too many tooltips—connected with many different text items in a text control— can be a burden and make your program seem difficult to use.

Separators

Separators are horizontal or vertical bars that let you organize your controls into groups. Although mostly used in menus to divide options into logical groupings, separators can also be used in `JApplet` and `JFrame` components, like any other control, as you'll see in this chapter.

That's it for the overview of what's in this chapter. As you can see, there's a lot coming up. It's time to turn to the "Immediate Solutions" section now, starting with combo boxes.

Immediate Solutions

Creating Combo Boxes

"Hmm," says the Novice Programmer, "I want to let the user select a word from a list of words, but now users say they want to enter their own words." "Sounds reasonable," you say. "What about using a combo box?" "OK," says the NP. "Can you add one to my code?"

Combo boxes can be used in two ways in Swing: as normal combo boxes and as drop-down lists. Drop-down lists let the users make a selection from a list that appears when they click a downwards pointing arrow. Normal combo boxes, on the other hand, are combinations of text fields and drop-down lists; the users can select an item from the drop-down list or enter their own text into the text field. Note that unlike list controls, only one item may be selected at a time in a combo box. Combo boxes are supported by the `JComboBox` class in Swing, and here's the inheritance diagram for that class:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.JcomboBox<E>
```

You'll find the fields for the `JComboBox` class in Table 14.1, its constructors in Table 14.2, and its methods in Table 14.3.

Table 14.1: Fields of the JComboBox class

Field	Does this
`protected String actionCommand`	The action command
`protected ComboBoxModel dataModel`	The data model
`protected ComboBoxEditor editor`	The editor responsible for the text field
`protected boolean isEditable`	It indicates whether the combo box is editable
`protected JComboBox.KeySelectionManager keySelectionManager`	The key selection manager
`protected boolean lightWeightPopupEnabled`	It indicates whether the drop-down list is enabled as a lightweight component
`protected int maximumRowCount`	It holds the row count
`protected ListCellRenderer<? super E> renderer`	The cell renderer
`protected Object selectedItemReminder`	A reminder for the selected item

Table 14.2: Constructors of the JComboBox class

Constructor	Does this
`JComboBox()`	It constructs a `JComboBox` object
`JComboBox(ComboBoxModel<E> aModel)`	It constructs a `JComboBox` object that takes items from an existing combo box model
`JComboBox(E[] items)`	It constructs a `JComboBox` object that contains the elements in the indicated array
`JComboBox(Vector<E> items)`	It constructs a `JComboBox` object that contains the elements in the indicated vector

Table 14.3: Methods of the JComboBox class

Method	Does this
`void actionPerformed (ActionEvent e)`	It adds an action
`void addActionListener (ActionListener l)`	It adds an action listener
`void addItem (E item)`	It adds an item to the item list
`void addItemListener (ItemListener aListener)`	It adds an item listener
`void configureEditor (ComboBoxEditor anEditor, Object anItem)`	It initializes the editor
`protected void configurePropertiesFromAction(Action a)`	It sets properties on this combobox to match those in specified `Action`
`void contentsChanged (ListDataEvent e)`	It is fired when contents in list are changed
`protected PropertyChangeListener createActionPropertyChangeListener(Action a)`	It creates and returns a `PropertyChangeListener` that is responsible for listening changes from the specified `Action` and updating the appropriate properties
`protected JComboBox.KeySelectionManager createDefaultKeySelectionManager()`	It gets an instance of the default key-selection manager
`protected void fireActionEvent()`	It notifies all listeners that have registered interest for notification on this event type
`protected void fireItemStateChanged(ItemEvent e)`	It notifies all listeners that have registered interest for notification on this event type
`AccessibleContext getAccessibleContext()`	It gets the accessible context
`Action getAction()`	It returns the currently set action for this `ActionEvent` source or null if no action is set
`String getActionCommand()`	It gets the action command
`ComboBoxEditor getEditor()`	It gets the editor used to edit the selected item in the `JComboBox` text field
`Object getItemAt(int index)`	It gets the list item at the indicated index
`int getItemCount()`	It gets the number of items in the list
`JComboBox.KeySelectionManager getKeySelectionManager()`	It gets the list's key-selection manager
`int getMaximumRowCount()`	It gets the maximum number of items the combo box can display at once
`ComboBoxModel getModel()`	It gets the data model
`ListCellRenderer getRenderer()`	It gets the renderer
`int getSelectedIndex()`	It gets the index of the currently selected item in the list
`Object getSelectedItem()`	It gets the currently selected item
`Object[] getSelectedObjects()`	It gets an array containing the selected item
`ComboBoxUI getUI()`	It gets the look and feel object that renders this component
`String getUIClassID()`	It gets the name of the look and feel class that renders this component
`void hidePopup()`	It causes the combo box to close its pop-up window

Table 14.3: Methods of the JComboBox class

Method	Does this
`void insertItemAt(Object anObject, int index)`	It inserts an item into the item list at a given index
`protected void installAncestorListener()`	It installs an ancestor listener
`void intervalAdded (ListDataEvent e)`	It is invoked when items have been added to the internal data model
`void intervalRemoved (ListDataEvent e)`	It is invoked when values have been removed from the data model
`boolean isEditable()`	It returns True if the `JComboBox` object is editable
`boolean isLightWeightPopupEnabled()`	It returns True if lightweight pop-ups are in use. It returns False if heavyweight pop-ups are used
`boolean isPopupVisible()`	It determines the visibility of the pop-up
`protected String paramString()`	It gets a string representation of this `JComboBox` object
`void processKeyEvent (KeyEvent e)`	It handles key events, looking for the Tab key
`void removeActionListener (ActionListener l)`	It removes an action listener
`void removeAllItems()`	It removes all items from the item list
`void removeItem(Object anObject)`	It removes an item from the item list
`void removeItemAt(int anIndex)`	It removes the item at `anIndex`. This method works only if the `JComboBox` object uses the default data model
`void removeItemListener (ItemListener aListener)`	It removes an item listener
`protected void selectedItemChanged()`	It is called when the selected item changes
`boolean selectWithKeyChar (char keyChar)`	It selects the list item that corresponds to the indicated keyboard character
`void setActionCommand (String aCommand)`	It sets the action command that should be included in the event sent to action listeners
`void setEditable (boolean aFlag)`	It sets the editable flag
`void setEditor (ComboBoxEditor anEditor)`	It sets the editor used to paint and edit the selected item in the `JComboBox` field
`void setEnabled (boolean b)`	It enables the combo box so that items can be selected
`void setKeySelectionManager (JComboBox.KeySelectionManager aManager)`	It sets the object that translates a keyboard character into a list selection
`void setLightWeightPopupEnabled (boolean aFlag)`	When displaying the pop-up, the `JComboBox` object chooses to use a lightweight pop-up if it fits
`void setMaximumRowCount (int count)`	It sets the maximum number of rows the `JComboBox` object displays
`void setModel (ComboBoxModel aModel)`	It sets the data model that the `JComboBox` object uses to obtain the list of items
`void setPopupVisible (boolean v)`	It sets the visibility of the pop-up
`void setRenderer (ListCellRenderer aRenderer)`	It sets the renderer that paints the item selected from the list in the `JComboBox` field
`void setSelectedIndex (int anIndex)`	It selects the item at index `anIndex`
`void setSelectedItem (Object anObject)`	It sets the selected item in the `JComboBox` object by specifying the object in the list

Table 14.3: Methods of the JComboBox class	
Method	**Does this**
`void setUI (ComboBoxUI ui)`	It sets the look and feel object that renders this component
`void showPopup()`	It causes the combo box to display its pop-up window
`void updateUI()`	It resets the UI property to a value from the current look and feel

The actual data in the combo box is stored in its model, which by default is an object of the `DefaultComboBoxModel` class. You can find the methods of this class in Table 14.4. Note that to add and delete items, you can use the model's methods, as shown in the table. To get the model object, you can use the `getModel()` method of `JComboBox`.

Table 14.4: Methods of the DefaultComboBoxModel class	
Method	**Does this**
`void addElement(EanObject)`	It adds an item to the end of the model
`E getElementAt(int index)`	It gets the value at the indicated index
`int getIndexOf(Object anObject)`	It gets the index position of the indicated object in the list
`Object getSelectedItem()`	It returns the selected item
`int getSize()`	It gets the length of the list
`void insertElementAt (E anObject, int index)`	It adds an item at a specific index
`void removeAllElements()`	It empties the list
`void removeElement(Object anObject)`	It removes an item from the model
`void removeElementAt(int index)`	It removes an item at a specific index
`void setSelectedItem (Object anObject)`	It sets the selected item

To add items to a combo box, you can use the `addItem()` method, passing it an object that it will pass on to its internal model. We put together an example here showing how this works. In this case, we just create a combo box and add six items—that is, six `String` objects—to it. Here's the code:

```
import java.awt.*;
import javax.swing.*;
/* <APPLET
        CODE = DemoJComboBox.class
        WIDTH = 200
        HEIGHT = 150 >
</APPLET>*/
public class DemoJComboBox extends JApplet {
        JComboBox combobox = new JComboBox();
        public void init() {
        Container c = getContentPane();
        combobox.addItem("Item 1");
        combobox.addItem("Item 2");
        combobox.addItem("Item 3");
        combobox.addItem("Item 4");
        combobox.addItem("Item 5");
        combobox.addItem("Item 6");
        c.setLayout(new FlowLayout());
        c.add(combobox); }
}
```

The result appears as shown in Figure 14.1, where you can see the combo box with its drop-down list open, displaying all six items.

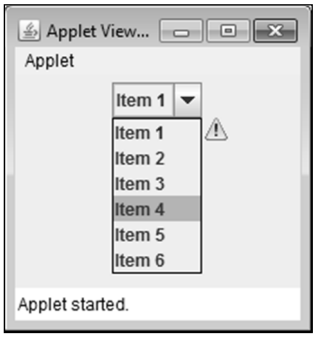


Figure 14.1: Creating a combo box

When the user selects an item from that list and releases the mouse button, that item becomes the newly selected item in the combo box, and it also appears in the text field. Note that, by default, the text field in the combo box is not editable—we'll show how to change that in a few pages.

So, how do you retrieve elements from a combo box? Because only one item in a combo box can be selected at one time, you can get that item with the `getSelectedItem()` method, which returns the item itself (note that the item is an object). You can also use `getSelectedIndex()` to determine the index of the currently selected item in the combo box's list, like this:

```
String item = (String) combobox.getSelectedItem();
int index = (int) combobox.getSelectedIindex();
```

To get an item at a specific index, use the model's `getElementAt()` method. You can also use the model's `insertElementAt()` and `removeElementAt()` methods to edit the list. Note that you can also respond to combo box events. We'll take a look at that in the next solution.

Handling Combo Box Selection Events

The Novice Programmer is back and says, "I want to change the drawing color in my program as soon as the user selects a new color from a combo box—how can I do that?" "It's easy," you say. "Just make your code respond to selection events." "Aha," says the NP, "here's my code, can you fix it?"

You can use two types of listeners with combo boxes—action listeners and item listeners. You use action listeners to handle editing events in the text field, and you use item listeners to handle list selections. We'll take a look at list selections in this solution and editing events in the next solution.

To handle selection events, we'll just add an item listener to the combo box example developed in the previous solution:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
        CODE = DemoJComboBoxEvent.class
        WIDTH = 250
        HEIGHT = 150 >
</APPLET>*/
public class DemoJComboBoxEvent extends JApplet implements ItemListener
{
        JComboBox combobox = new JComboBox();
        String msg = "";
        public void init()
        {
                Container c = getContentPane();
                combobox.addItem("Item 1");
                combobox.addItem("Item 2");
                combobox.addItem("Item 3");
                combobox.addItem("Item 4");
                combobox.addItem("Item 5");
```

```
                combobox.addItem("Item 6");
                c.setLayout(new FlowLayout());
                c.add(combobox);
                combobox.addItemListener(this);
        }
        . . .
    }
```

Now we add the `itemStateChanged()` method to this applet that the `ItemListener` interface needs:

```
public void itemStateChanged(ItemEvent e) { . . . }
```

This method is passed an object of the `ItemEvent` class, and we can determine whether the corresponding item in the combo box list was selected by using the `getStateChange()` method, like this:

```
public void itemStateChanged(ItemEvent e) {
        if(e.getStateChange() == ItemEvent.SELECTED)
        . . .
    }
```

We'll indicate whether an item was selected or deselected in the status bar of the applet like this:

```
public void itemStateChanged(ItemEvent e) {
        if(e.getStateChange() == ItemEvent.SELECTED)
                msg += " Selected: " + (String)e.getItem();
        else
                msg += " Deselected: " + (String)e.getItem();
        showStatus(msg);

}
```

The result of this code appears as shown in Figure 14.2:



Figure 14.2: Working with combo box selection events

As you can see, at the bottom of the applet in the figure the code reports every time the user deselects one item and selects another.

Creating Editable Combo Boxes

"Oh no!" says the NP. "I can't get the text field in my combo box to work—I can't seem to enter any text in it." You smile and say, "That's because you have to make it *editable*." "Oh!" says the NP. "How do you do that?"

To make the text field in a combo box editable, you use the `setEditable()` method. To handle the editing events, you use an action listener. Here's an example where we make a combo box editable and add an action listener to it:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
        CODE = DemoJComboBoxEdit.class
        WIDTH = 200
        HEIGHT = 150 >
</APPLET>*/
```

```
public class DemoJComboBoxEdit extends JApplet implements ActionListener {
  private JComboBox combobox = new JComboBox();
        public void init() {
  Container c = getContentPane();
  combobox.addItem("Item 1");
  combobox.addItem("Item 2");
  combobox.addItem("Item 3");
  combobox.addItem("Item 4");
  combobox.addItem("Item 5");
  combobox.addItem("Item 6");
  combobox.setEditable(true);
  c.setLayout(new FlowLayout());
  c.add(combobox);
  combobox.getEditor().addActionListener(this); }
. . .
}
```

Note that we are not adding an action listener to the combo box directly but rather to the *editor* in the combo box, which is an object that implements the `ComboBoxEditor` interface. You get this object with the `getEditor()` method. The editor is responsible for the editing actions in the text field in the combo box, and you'll find the methods of `ComboBoxEditor` in Table 14.5:

Table 14.5: Methods of the ComboBoxEditor interface

Method	Does this
`void addActionListener (ActionListener l)`	It adds an action listener
`Component getEditorComponent()`	It returns the editor component
`Object getItem()`	It returns the edited item
`void removeActionListener (ActionListener l)`	It removes an action listener
`void selectAll()`	It asks the editor to start editing and select everything
`void setItem(Object anObject)`	It sets the item that should be edited

In this example, we display the item's text before it's edited as well as after it has been edited, and the user presses Enter. To get the item's text before it's edited, we use the `getSelectedItem()` method; and to get the item after it has been edited, we use the editor's `getItem()` method, like this:

```
public void actionPerformed(ActionEvent e) {
        String msg = (String)combobox.getSelectedItem()
        + " was changed to " + (String)combobox.getEditor().getItem();
        showStatus(msg);
}
```

The result appears as shown in Figure 14.3:



Figure 14.3: Editing the text field in a combo box

When the user edits an item in the text field and presses Enter, the old and new text in the combo box appears in the applet's status bar, as you can see in the figure.

Adding Images to Combo Boxes

"Say," says the NP, "isn't there any way to display images in a combo box?" "There sure is," you say. "Just sit down and we'll take a look at it."

You can add images to combo boxes just as you can to list controls. In fact, when you add images to combo boxes, you actually add images to the list part of the combo box, so the process is much the same as adding images to lists.

Here's an example in which we create a new model, `MyModel`, and cell renderer, `MyRenderer`, for a combo box. Here's how we add them in the combo box:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
        CODE = DemoJComboBoxImage.class
        WIDTH = 400
        HEIGHT = 600 >
</APPLET>*/
public class DemoJComboBoxImage extends JApplet {
        public void init() {
  Container c = getContentPane();
  c.setLayout(new FlowLayout());
  MyModel model = new MyModel();
  MyRenderer renderer = new MyRenderer();
  JComboBox combobox = new JComboBox(model);
  combobox.setRenderer(renderer);
  c.add(new JScrollPane(combobox)); }
}
```

This new model and renderer will store and display images in the combo box, and the result appears as shown in Figure 14.4:

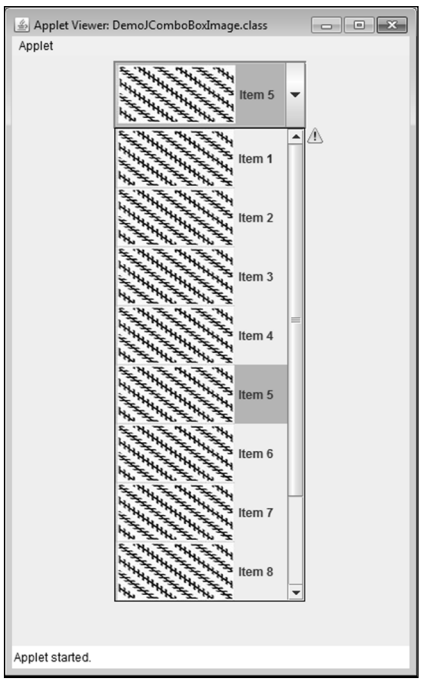


Figure 14.4: Adding images to combo boxes

Now all that remains is to create the new model and renderer, and we'll do that in the next two solutions.

Creating a Combo Box Model

We'll create a new combo box model in this solution for use with the combo box in the previous solution (the one that displays images). This new model, which extends the `DefaultComboBoxModel` class, will use the `addElement()` method to store data elements in the model. Each element is made up of a text string and an image icon, like this:

```
class MyModel extends DefaultComboBoxModel
{
  public MyModel()
  {
        for(int loop_index = 1; loop_index <= 10; loop_index++)
        {
                addElement(new Object[] {"Item " + loop_index,
                new ImageIcon("zoll1.jpg")});
        }
  }
}
```

To put this model to work, see the next topic.

Creating a Combo Box Custom Renderer

In this solution, we'll create a renderer that displays images in a combo box (this renderer is for use in the code in the previous two solutions). To create the cell renderer for a combo box, you can just extend the `JLabel` class, which is the class of each item in a combo box, and implement the `ListCellRenderer` interface. To implement that interface, we override the `getListCellRenderer()` method by using the `JLabel` class's `setText()` and `setIcon()` methods to add text and images in the combo box:

```
class MyRenderer extends JLabel implements ListCellRenderer
{
  public MyRenderer() { setOpaque(true);
}
  public Component getListCellRendererComponent(
  JList list, Object obj, int index, boolean isSelected,
  boolean focus) {
        MyModel model = (MyModel)list.getModel();
        setText((String)((Object[])obj)[0]);
        setIcon((Icon)((Object[])obj)[1]);
        if(!isSelected) {
        setBackground(list.getBackground());
        setForeground(list.getForeground()); }
                else {
        setBackground(list.getSelectionBackground());
        setForeground(list.getSelectionForeground()); }
        return this; }
}
```

The result appears in Figure 14.4, where you can see the images displayed in the combo box.

Creating Progress Bars

"Well," the NP says, "it's true that it takes three hours for my program to sort its internal data, but is that the reason for users to complain?" "It surely is," you say, "unless you give them some kind of visual indication of what's going on. How about adding a progress bar to your program." "Sure!" says the NP, "What's a progress bar?"

A progress bar control draws an updated colored bar that displays the progress of some operation. You can set the minimum and maximum values for a progress bar when you call the `JProgressBar` constructor, or you can use the `setMinimum()` and `setMaximum()` methods. You can then repeatedly update the progress bar by calling its `setValue()` method to indicate the progressive status of an operation. Here's the inheritance diagram for the `JProgressBar` class:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
```

```
                    |____javax.swing.JComponent
                          |____javax.swing.JProgressBar
```

You can find the fields of the `JProgressBar` class in Table 14.6 (note that it also inherits two constants from `SwingConstants` that you use for orientation: `HORIZONTAL` and `VERTICAL`), its constructors in Table 14.7, and its methods in Table 14.8.

Table 14.6: Fields of the JProgressBar class

Field	Does this
`protected ChangeEvent changeEvent`	The change event
`protected ChangeListener changeListener`	The change listener
`protected BoundedRangeModel model`	The data model that holds data values for the progress bar
`protected int orientation`	The orientation in which to display the progress bar
`protected boolean paintBorder`	It returns True if there should be a border around the progress bar
`protected boolean paintString`	It returns True if there should be a string label in the progress bar
`protected String progressString`	An optional string that can be displayed on the progress bar

Table 14.7: Constructors of the JProgressBar class

Constructor	Does this
`JProgressBar()`	It constructs a horizontal progress bar (the default orientation) that displays a border but no progress string
`JProgressBar(BoundedRangeModel newModel)`	It constructs a horizontal progress bar (the default orientation) to hold the data using the specified model
`JProgressBar(int orient)`	It constructs a progress bar with the indicated orientation, which can be either `JProgressBar.VERTICAL` or `JProgressBar.HORIZONTAL`
`JProgressBar (int min, int max)`	It constructs a horizontal progress bar using the specified minimum and maximum values
`JProgressBar(int orient, int min, int max)`	It constructs a progress bar using the indicated orientation, minimum, and maximum values

Table 14.8: Methods of the JProgressBar class

Method	Does this
`void addChangeListener (ChangeListener l)`	It adds a change listener
`protected ChangeListener createChangeListener()`	It creates a change listener
`protected void fireStateChanged()`	It notifies all listeners that have registered interest for notification on this event type
`AccessibleContext getAccessibleContext()`	It gets the accessible context associated with this `JProgressBar` object
`ChangeListener[] getChangeListeners()`	It gets an array of all the `ChangeListener`s added to this progress bar with `addChangeListener()`.
`int getMaximum()`	It gets the model's maximum value
`int getMinimum()`	It gets the model's minimum value
`BoundedRangeModel getModel()`	It gets the data model
`int getOrientation()`	It returns `JProgressBar.VERTICAL` or `JProgressBar.HORIZONTAL`
`double getPercentComplete()`	It gets the percentage/percent complete for the progress bar
`String getString()`	It gets the current value of the progress string

Table 14.8: Methods of the JProgressBar class

Method	Does this
`ProgressBarUI getUI()`	It gets the look and feel object that renders this component
`String getUIClassID()`	It gets the name of the look and feel class that renders this component
`int getValue()`	It gets the model's current value
`boolean isBorderPainted()`	It returns True if the progress bar has a border and false if it does not
`boolean isStringPainted()`	It returns True if a string will be drawn in the progress bar
`protected void paintBorder (Graphics g)`	It paints the progress bar's border (as long as the `borderPainted` property is True)
`protected String paramString()`	It gets a string representation of the `JProgressBar` object
`void removeChangeListener (ChangeListener l)`	It removes a change listener from the button
`void setBorderPainted (boolean b)`	It sets whether the progress bar should paint its border
`void setIndeterminate(boolean newValue)`	It sets the indeterminate property of the progress bar is in determinate or indeterminate mode.
`void setMaximum(int n)`	It sets the model's maximum value to `x`
`void setMinimum(int n)`	It sets the model's minimum value to `x`
`void setModel (BoundedRangeModelnewModel)`	It sets the data model used by the `JProgressBar` object
`void setOrientation (int newOrientation)`	It sets the progress bar's orientation to `newOrientation`
`void setString(String s)`	It sets the value of the progress string
`void setStringPainted (boolean b)`	It specifies whether the progress bar will render a string
`void setUI (ProgressBarUI ui)`	It sets the look and feel object that renders this component
`void setValue(int n)`	It sets the model's current value to `n`
`void updateUI()`	It is called by the `UIFactory` class to indicate that the look and feel has changed

There are quite a few ways to display progress bars. For example, you can make them horizontal or vertical when you call the class's constructor or use the `setOrientation()` method. You can also select the drawing color for the actual bar, display a label inside the progress bar that indicates the current value, and more. The basic use of all progress bars, however, is the same no matter what they look like—you set a minimum and maximum value for the progress bar (the default values are 0 and 100) and then update its display with the `setValue()` method, as appropriate. Here are a few examples showing some of the ways you can display progress bars:

```
import java.awt.*;
import javax.swing.*;
/* <APPLET
        CODE = DemoJProgressBar.class
        WIDTH = 400
        HEIGHT = 200 >
</APPLET>*/
public class DemoJProgressBar extends JApplet
{
  JProgressBar bar1, bar2, bar3,bar4, bar5, bar6;
  public void init()
  {
        Container c= getContentPane();
        c.setLayout(new FlowLayout());
        bar1 = new JProgressBar();
```

```
            bar1.setValue(40);
            c.add(bar1);
            bar2 = new JProgressBar();
            bar2.setMinimum(70);
            bar2.setMaximum(150);
            bar2.setValue(120);
            bar2.setForeground(Color.green);
            c.add(bar2);
            bar3 = new JProgressBar();
            bar3.setOrientation(JProgressBar.VERTICAL);
            bar3.setForeground(Color.red);
            bar3.setValue(30);
            bar3.setStringPainted(true);
            bar3.setBorder(BorderFactory.createRaisedBevelBorder());
            c.add(bar3);
            bar4 = new JProgressBar();
            bar4.setOrientation(JProgressBar.VERTICAL);
            bar4.setForeground(Color.blue);
            bar4.setValue(70);
            bar4.setStringPainted(true);
            bar4.setBorderPainted(false);
            c.add(bar4);
            bar5 = new JProgressBar();
            bar5.setOrientation(JProgressBar.VERTICAL);
            bar5.setStringPainted(true);
            bar5.setString("Hello from Java!");
            bar5.setValue(80);
            c.add(bar5);
        }
    }
```

The result of this code appears as shown in Figure 14.5, where you can see various progress bars displayed in a number of ways: There is a lot of possibilities here (note, in particular, that you can add a label to the progress bar to indicate its current setting, and you can customize the progress bar's border).

On the other hand, static progress bars, like the one shown in Figure 14.5, aren't much used because they don't do anything:

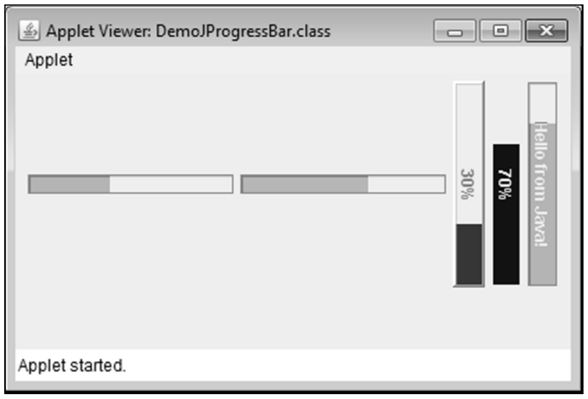


Figure 14.5: Displaying progress bars

To see how to update a progress bar, take a look at the next solution.

Updating Progress Bars

The NP says, "Gah! I put progress bars into my program, but they're not doing anything—what's wrong?" "Well," you say, "that's because you have to call the `setValue()` method yourself to update them." "Oh," says the NP.

After creating and displaying a progress bar, it's up to you to update it with the `setValue()` method. You'll usually do that by dividing your time-consuming task into parts, and you'll update the progress bar as each part completes:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
        CODE = DemoJProgressBarUpdate.class
        WIDTH = 400
        HEIGHT = 200 >
</APPLET>*/
public class DemoJProgressBarUpdate extends JApplet {
        JProgressBar bar = new JProgressBar();
        JButton button = new JButton("Increment the progress bar");
        public void init() {
        Container c= getContentPane();
        c.setLayout(new FlowLayout());
        c.add(bar);
        c.add(button);
        bar.setStringPainted(true);
        button.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
  bar.setValue(bar.getValue() + 15);    }});
  }
}
```

The result appears as shown in Figure 14.6:

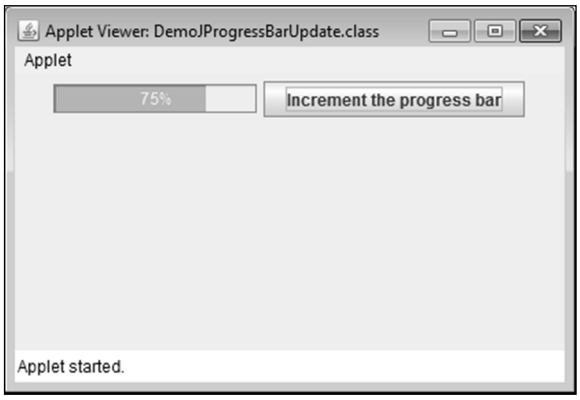


Figure 14.6: Updating a progress bar

Note that you can update the progress bar just by clicking the button. Usually, of course, your code will update the progress bar itself because progress bars are used to indicate the progress of time-consuming operations to the user.

Handling Progress Bar Events

Progress bar events occur every time the progress bar's value changes, and you can work with change listeners to catch those events. It catches the events that occur as they update the value of the progress bar. In this case, we report the new value of the progress bar each time it changes as well as its minimum and maximum values, like this:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;

/* <APPLET
```

```
            CODE = DemoJProgressBarEvent.class
            WIDTH = 400
            HEIGHT = 200 >
    </APPLET>*/
    public class DemoJProgressBarEvent extends JApplet
    {
       JProgressBar bar = new JProgressBar();
       JButton button = new JButton("Increment the progress bar");
       public void init()
       {
            Container c = getContentPane();
            c.setLayout(new FlowLayout());
            c.add(button);
            bar.setForeground(Color.pink);
            c.add(bar);
            button.addActionListener(new ActionListener()
            {
                    public void actionPerformed(ActionEvent e)
                    {
                    bar.setValue(bar.getValue() + 15); }});
                    bar.addChangeListener(new ChangeListener()
                    {
                            public void stateChanged(ChangeEvent e)
                            {
                                    showStatus("Minimum: " + bar.getMinimum()
                                    + " Maximum: " + bar.getMaximum() +
                                    " Value: " + bar.getValue()); }} );
            }
    }
```

The result of this code appears as shown in Figure 14.7:

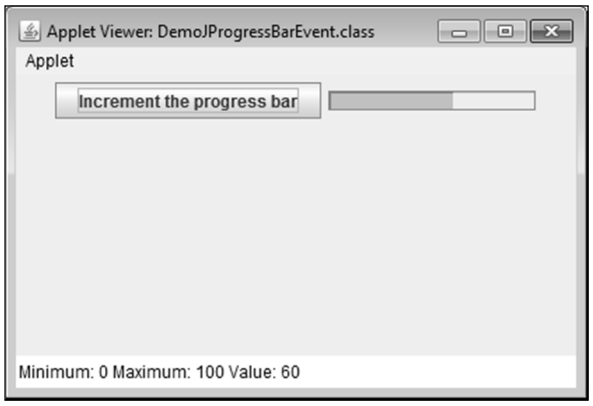


Figure 14.7: Handling progress bar events

As you can see in the figure, each time the user updates the progress bar, its new value appears in the applet's status bar.

Sometimes, you don't know how long a task is or it may happen that the task may get stuck in the same state for quite some time. To handle such situations, Java allows you to put the progress bar in the indeterminate mode. In this mode, an animating progress bar indicates whether the task is functional or not. You can switch to the default determinate mode when you feel that the progress bar can display more meaningful information.

Creating Tooltips

"Say," says the Product Support Specialist, "no one seems to read the manuals, ever. Isn't there some way to make our programs self-explanatory?" "Well," you say, "we could reduce everything to a series of cartoons." "Hmm," says the PSS thoughtfully. "Just kidding," you say quickly. "I'll add tooltips to the controls in the program."

Tooltips are small windows that appear when the user lets the mouse rest on a component. They display some text that explains the purpose of the control. For example, a Cut button's tooltip might read "Cut selected text." Tooltips are supported with the Swing `JToolTip` class, and here's that class's inheritance diagram:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.JToolTip
```

You'll find the constructor of the `JToolTip` class in Table 14.9 and its methods in Table 14.10:

Table 14.9: The constructor of the JToolTip class

Constructor	Does this
`JToolTip()`	It constructs a tooltip

Table 14.10: Methods of the JToolTip class

Method	Does this
`AccessibleContext getAccessibleContext()`	It gets the accessible context
`JComponent getComponent()`	It gets the component the tooltip applies to
`String getTipText()`	It gets the text that's shown in the tooltip
`ToolTipUI getUI()`	It gets the look and feel object that renders this component
`String getUIClassID()`	It gets the name of the look and feel class that renders this component
`protected String paramString()`	It gets a string representation of this `JToolTip` object
`void setComponent(JComponent c)`	It sets the component that the tooltip describes
`void setTipText(String tipText)`	It sets the text shown when the tooltip appears
`void updateUI()`	It is called by the `UIFactory` class when the look and feel has changed

In fact, many components already have a method, `setToolTipText()`, that you can use to add a tooltip. Here's an example in which we add a tooltip to a button:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
         CODE=DemoToolTip.class
         WIDTH=300
         HEIGHT=200 >
</APPLET>*/
public class DemoToolTip extends JApplet {
         JButton button = new JButton("Click Here");
         JTextField text = new JTextField(15);
         public void init() {
         Container c = getContentPane();
         c.setLayout(new FlowLayout());
         button.setToolTipText("This is a button.");
         text.setToolTipText("This is a text field.");
         c.add(button);
         c.add(text);
         button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent event) {
         text.setText("Kogent Solutions"); }});
         }
}
```

It's as easy as adding a tooltip to most components. The result appears as shown in Figure 14.8, where you can see the tooltip with its explanatory text as "This is a button":

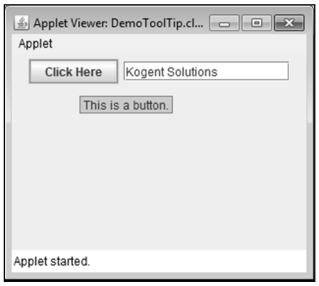


Figure 14.8: Using tooltips

For the components with multiple parts, such as tabbed panes, you can reflect the part of component under the cursor using varied tooltip text. For example, you can give information about what will happen when you click the tabbed pane in a tooltip using the `setToolTipText()` method or by passing the tooltip text as an argument in the `addTab()` method. For components with no API to set the tooltip text, you can set the tooltip text using a custom renderer if the component supports renderers.

Creating Separators

The NP says, "The Big Boss told me that there are too many text fields in my program and I should divide them into groups." "You can do that in several easy ways," you say, "such as adding text fields to panels and giving the panels various background colors. You can even use separators. By the way, how many text fields are there in your program?" "About 2,413," the NP says. "Uh-oh," you say.

Separators are horizontal or vertical lines, and they usually appear in menus to separate options into logical groups, but you can also use them to separate components in a layout. They're supported by the `JSeparator` class in Swing, and here's that class's inheritance diagram:

```
java.lang.Object
   |____java.awt.Component
         |____java.awt.Container
               |____javax.swing.JComponent
                     |____javax.swing.JSeparator
```

You can find the constructors of the `JSeparator` class in Table 14.11 and its methods in Table 14.12:

Table 14.11: Constructors of the JSeparator class

Constructor	Does this
`JSeparator()`	It creates a new horizontal separator
`JSeparator(int orientation)`	It creates a new separator with the indicated horizontal or vertical orientation

Table 14.12: Methods of the JSeparator class

Method	Does this
`AccessibleContext getAccessibleContext()`	It gets the accessible context
`int getOrientation()`	It gets the orientation of this separator
`SeparatorUI getUI()`	It gets the look and feel object that renders this component
`String getUIClassID()`	It gets the name of the look and feel class that renders this component
`protected String paramString()`	It gets a string representation of this `JSeparator` object
`void setOrientation(int orientation)`	It sets the orientation of the separator
`void setUI(SeparatorUI ui)`	It sets the look and feel object that renders this component
`void updateUI()`	It is called by the `UIFactory` class when the look and feel has changed

Let's look at an example. Here, we put a separator between two text fields in a flow layout. To make a separator visible, you have to do more than just add it to a layout —you also have to give it a preferred size by using its `setPreferredSize()` method. The usual way to do that is to use the separator's `getPreferredSize()` method to get the current width of the separator and then pass that and the new length you want to the `setPreferredSize()` method. The `setPreferredSize()` and `getPreferredSize()` methods work with objects of the AWT `Dimension` class, which has two fields: `width` and `height`. Here's how we create the new separator and get its dimensions:

```
import java.awt.*;
import javax.swing.*;
/*<APPLET
         CODE = DemoJSeparator.class
         WIDTH = 300
         HEIGHT = 200 >
</APPLET>*/
public class DemoJSeparator extends JApplet {
         JSeparator separator = new JSeparator(JSeparator.VERTICAL);
         Dimension dim = separator.getPreferredSize();
         . . .
```

Now we can put the separator between two text fields, like this, making it 100 pixels high:

```
public class DemoJSeparator extends JApplet {
         JSeparator separator = new JSeparator(JSeparator.VERTICAL);
         Dimension dim = separator.getPreferredSize();
         public void init() {
         Container c = getContentPane();
         c.setLayout(new FlowLayout());
         c.add(new JTextField("Kogent Solutions"));
         c.add(separator);
         c.add(new JTextField("Kogent Solutions"));
         separator.setPreferredSize(new Dimension(dim.width, 100));}
}
```

The result of this code appears as shown in Figure 14.9:

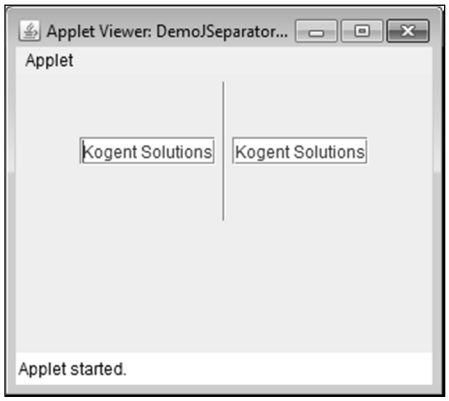


Figure 14.9: Creating a separator

As you can see in the figure, the separator appears between the two text fields.

There's one problem here, though—what if the user resizes the applet? In that case, you might want to resize the separator to match. You can do that by handling resizing events—see the next solution for the details.

Resizing Separators Automatically

To resize a separator, you can catch component resizing events by using the `ComponentListener` interface, which handles the resizing events for components, including applets and windows. The methods of this interface appear in Table 14.13.

Table 14.13: Methods of the ComponentListener interface	
Method	**Does this**
`void componentHidden (ComponentEvent e)`	It is called when the component is made invisible
`void componentMoved (ComponentEvent e)`	It is called when the component's position changes
`void componentResized (ComponentEvent e)`	It is called when the component's size changes
`void componentShown (ComponentEvent e)`	It is called when the component is made visible

For example, when you resize an applet, you can resize a separator to the new size you want. Here's an example in which we create a separator between two text fields that goes from the top of an applet window's client area to the bottom and maintains that extent even when the applet is resized. Here's what the code looks like—note the resizing code in the `ComponentShown` (to display the separator the first time) and `ComponentResized` methods:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
        CODE = DemoJSeparatorEvent.class
        WIDTH = 300
        HEIGHT = 200 >
</APPLET>*/
public class DemoJSeparatorEvent extends JApplet implements ComponentListener {
        JSeparator separator = new JSeparator(JSeparator.VERTICAL);
        Dimension dim = separator.getPreferredSize();
        public void init() {
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        c.add(new JTextField("Kogent Solutions"));
        c.add(separator);
        c.add(new JTextField("Kogent Solutions"));
        addComponentListener(this); }
        public void componentShown(ComponentEvent e)    {
        separator.setPreferredSize(new Dimension(dim.width,getSize().height));
        separator.revalidate(); }
        public void componentResized(ComponentEvent e) {
        separator.setPreferredSize(new Dimension(dim.width,getSize().height));
        separator.revalidate(); }
        public void componentMoved(ComponentEvent e) { }
        public void componentHidden(ComponentEvent e) { }
}
```

The result of this code appears as shown in Figure 14.10:

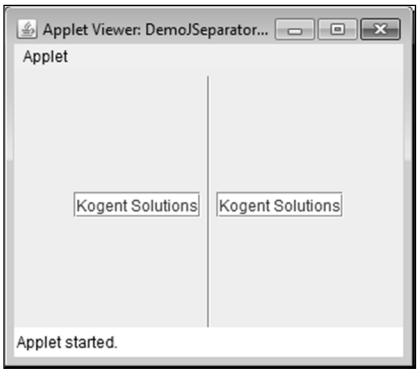


Figure 14.10: Creating a separator that resizes itself

As you can see in the figure, the separator extends from the top of the client area in the applet to the bottom—and it'll resize itself as the applet does.

Creating Color Choosers

"Now I've got a real problem," the NP says. "I want to let the users select a drawing color for my new program, *SuperDuperArtisticPro*, but I can't ask them to just type in new color values." "Why not use a color chooser?" you ask. "Good idea!" the NP says. "You write the code and I'll watch."

You can let the user select a color with the Swing `JColorChooser` class, which is a prebuilt dialog box that presents the user with several ways of choosing colors. Here's the inheritance diagram for `JColorChooser`:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.Jcomponent
                    |____javax.swing.JColorChooser
```

You can find the fields of the `JColorChooser` class in Table 14.14, its constructors in Table 14.15, and its methods in Table 14.16.

Table 14.14: Fields of the JColorChooser class

Field	Does this
`protected AccessibleContext accessibleContext`	The accessible context
`static String CHOOSER_PANELS_PROPERTY`	The chooser panel array property name
`static String PREVIEW_PANEL_PROPERTY`	The preview panel property name
`static String SELECTION_MODEL_PROPERTY`	The selection model property name

Table 14.15: Constructors of the JColorChooser class

Constructor	Does this
`JColorChooser()`	It constructs a color chooser pane
`JColorChooser(Color initialColor)`	It constructs a color chooser pane with the indicated initial color
`JColorChooser(ColorSelectionModel model)`	It constructs a color chooser pane with the indicated color-selection model

Table 14.16: Methods of the JColorChooser class

Method	Does this
`void addChooserPanel (AbstractColorChooserPanel panel)`	It adds a color chooser panel
`static JDialog createDialog (Component c, String title, boolean modal, JColorChooser chooserPane, ActionListener okListener, ActionListener cancelListener)`	It constructs and returns a new dialog box with the indicated color chooser and with OK, Cancel, and Reset buttons
`AccessibleContext getAccessibleContext()`	It gets the accessible context
`AbstractColorChooserPanel[] getChooserPanels()`	It gets the indicated color panels
`Color getColor()`	It gets the current color value from the color chooser
`JComponent getPreviewPanel()`	It gets the preview panel that shows a chosen color
`ColorSelectionModel getSelectionModel()`	It gets the data model that handles color selections
`ColorChooserUI getUI()`	It gets the look and feel object that renders this component

Table 14.16: Methods of the JColorChooser class

Method	Does this
`String getUIClassID()`	It gets the name of the look and feel class that renders this component
`protected String paramString()`	It gets a string representation of this `JColorChooser` object
`AbstractColorChooserPanel removeChooserPanel (AbstractColorChooserPanel panel)`	It removes the color panel indicated
`void setChooserPanels (AbstractColorChooserPanel [ ] panels)`	It specifies the color panels used to choose a color value
`void setColor(Color color)`	It sets the current color of the color chooser to the indicated color
`void setColor(int c)`	It sets the current color of the color chooser to the indicated color
`void setColor(int r, int g, int b)`	It sets the current color of the color chooser to the indicated RGB color
`void setDragEnabled (boolean b)`	It sets the drag enabled property
`void setPreviewPanel (JComponent preview)`	It sets the current preview panel
`void setSelectionModel (ColorSelectionModel newModel)`	It sets the model containing the selected color
`void setUI (ColorChooserUI ui)`	It sets the look and feel object that renders this component
`static Color showDialog(Component component, String title, Color initialColor)`	It shows a modal color chooser dialog box
`void updateUI()`	It is called by the `UIManager` class when the look and feel has changed

Using a color chooser is easy; all you have to do is to show it with its `showDialog()` method, passing that method a parent object, a caption, and a default color. This method returns the color selected by the user as a `Color` object (or the default color if the user didn't select a color).

Let's look at an example in which we place a button in a panel and let the user display a color chooser when the button is clicked. After the user selects a color, the code will set the background of the panel to that color. Here's what the code looks like:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
        CODE = DemoJColorChooser.class
        WIDTH = 200
        HEIGHT = 200 >
</APPLET>*/
public class DemoJColorChooser extends JApplet implements ActionListener {
        JPanel panel = new JPanel();
        JButton button;
        public void init() {
        button = new JButton("Change background color");
        button.addActionListener(this);
        panel.add(button);
        getContentPane().add(panel); }
        public void actionPerformed(ActionEvent e) {
        Color color = JColorChooser.showDialog(DemoJColorChooser.this,
        "Select the color of your choice", Color.white);
```

```
            panel.setBackground(color); }
    }
```

As you can see, it's easy to use a color chooser; the color chooser appears when the user clicks the Change background color button, as shown in Figure 14.11 (a).

When the user selects a color in the color chooser, the new color appears in the applet's panel, as shown in Figure 14.11 (b).



Figure 14.11: (a) Creating a color chooser

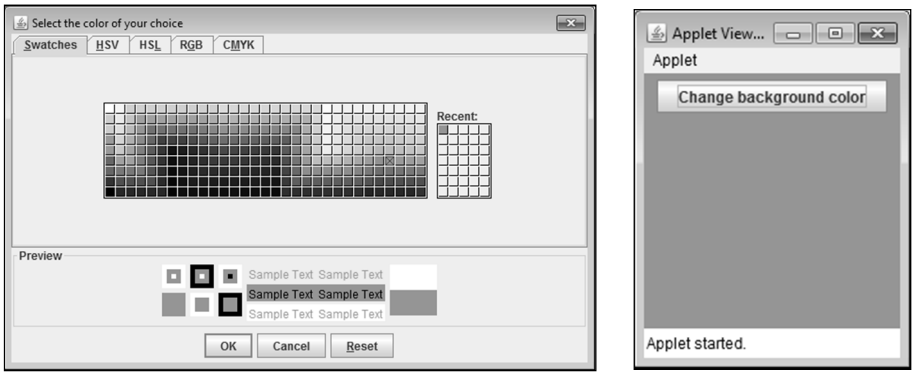


Figure 14.11(b): Changing the color of a panel to the color selected

Creating File Choosers

"Hmm," says the NP, "I need to get the name of a file from the user—is there any quick way of doing that?" "Sure," you say. "A text file." "Well," the NP says, "I want to let the user browse for the correct file, if needed." "Oh," you say, "then you want a file chooser."

File choosers are dialog boxes that let users specify the name and path of a file, and browse their disk storage if they need to. File choosers are supported by the `JFileChooser` class. Here's the inheritance diagram for that class:

```
java.lang.Object
   |____java.awt.Component
         |____java.awt.Container
               |____javax.swing.JComponent
                     |____javax.swing.JFileChooser
```

You'll find the fields of the `JFileChooser` class in Table 14.17, its constructors in Table 14.18, and its methods in Table 14.19:

Table 14.17: Fields of the JFileChooser class

Field	Does this
`static String ACCEPT_ALL_FILE_FILTER_USED_CHANGED_PROPERTY`	It identifies whether the `AcceptAllFileFilter` is used or not.
`protected AccessibleContext accessibleContext`	The accessible context
`static String ACCESSORY_CHANGED_PROPERTY`	It indicates that a different accessory component is in use
`static String APPROVE_BUTTON_MNEMONIC_CHANGED_PROPERTY`	It indicates a change in the mnemonic for the Yes/OK button
`static String APPROVE_BUTTON_TEXT_CHANGED_PROPERTY`	It indicates a change in the text on the Yes/OK button
`static String APPROVE_BUTTON_TOOL_TIP_TEXT_CHANGED_PROPERTY`	It indicates a change in the tooltip text for the Yes/OK button
`static int APPROVE_OPTION`	The return value if the Yes/OK button is chosen
`static String APPROVE_SELECTION`	This is a instruction to approve the current selection
`static int CANCEL_OPTION`	The return value if the Cancel button is chosen
`static String CANCEL_SELECTION`	This is a instruction to cancel the current selection
`static String CHOOSABLE_FILE_FILTER_CHANGED_PROPERTY`	It indicates a change in the list of predefined file filters the user can choose from
`static String CONTROL_BUTTONS_ARE_SHOWN_CHANGED_PROPERTY`	It is an instruction to display the control buttons.
`static int CUSTOM_DIALOG`	It is a type value indicating that the file chooser supports a developer-specified file operation
`static String DIALOG_TITLE_CHANGED_PROPERTY`	It indicates a change in the dialog box's title
`static String DIALOG_TYPE_CHANGED_PROPERTY`	It indicates a change in the type of files displayed (files only, directories only, or both files and directories)
`static int DIRECTORIES_ONLY`	This is a instruction to display only directories
`static String DIRECTORY_CHANGED_PROPERTY`	It indicates a directory change
`static int ERROR_OPTION`	The return value if an error occurred
`static String FILE_FILTER_CHANGED_PROPERTY`	It indicates the user changed the kind of files to display
`static String FILE_HIDING_CHANGED_PROPERTY`	It indicates a change in the display-hidden-files property
`static String FILE_SELECTION_MODE_CHANGED_PROPERTY`	It indicates a change in the kind of selection (single, multiple, and so on)
`static String FILE_SYSTEM_VIEW_CHANGED_PROPERTY`	It indicates that a different object is being used to find available drives on the system
`static String FILE_VIEW_CHANGED_PROPERTY`	It indicates that a different object is being used to retrieve file information
`static int FILES_AND_DIRECTORIES`	This is an instruction to display both files and directories
`static int FILES_ONLY`	This is an instruction to display only files
`static String MULTI_SELECTION_ENABLED_CHANGED_PROPERTY`	It enables multiple-file selections
`static int OPEN_DIALOG`	It indicates that the file chooser supports an "Open" file operation

Table 14.17: Fields of the JFileChooser class

Field	Does this
`static int SAVE_DIALOG`	It indicates that the file chooser supports a "Save" file operation
`static String SELECTED_FILE_CHANGED_PROPERTY`	It indicates a change in the user's single-file selection
`static String SELECTED_FILES_CHANGED_PROPERTY`	It indicates a change in the user's multiple-file selection

Table 14.18: Constructors of the JFileChooser class

Constructor	Does this
`JFileChooser()`	It constructs a `JFileChooser` object
`JFileChooser(File currentDirectory)`	It constructs a `JFileChooser` object using the given file as the path
`JFileChooser(File currentDirectory, FileSystemView fsv)`	It constructs a `JFileChooser` object using the given current directory and file system view
`JFileChooser(FileSystemView fsv)`	It constructs a `JFileChooser` object using the given file system view
`JFileChooser(String currentDirectoryPath)`	It constructs a `JFileChooser` object using the given path
`JFileChooser(String currentDirectoryPath, FileSystemView fsv)`	It constructs a `JFileChooser` object using the given current directory path and file system view

Table 14.19: Methods of the JFileChooser class

Method	Does this
`Boolean accept(File f)`	It returns True if the file should be displayed
`void addActionListener (ActionListener l)`	It adds an action listener
`void addChoosableFileFilter (FileFilter filter)`	It adds a filter to the list of choosable file filters
`void approveSelection()`	It is called by the UI when the user clicks the Save or Open button
`void cancelSelection()`	It is called by the UI when the user clicks the Cancel button
`void changeToParentDirectory()`	It changes the directory to be set to the parent of the current directory
`void ensureFileIsVisible (File f)`	It makes sure that the indicated file is visible
`protected void fireActionPerformed(String command)`	It notifies all listeners that have registered interest for notification on this event type
`FileFilter getAcceptAllFileFilter()`	It gets the `AcceptAll` file filter
`AccessibleContext getAccessibleContext()`	It gets the accessible context associated with this `JFileChooser` object
`JComponent getAccessory()`	It gets the accessory component
`int getApproveButtonMnemonic()`	It gets the approve button's mnemonic
`String getApproveButtonText()`	It gets the text used in the approve button in the `FileChooserUI`
`String getApproveButtonToolTipText()`	It gets the tooltip text used in the approve button
`FileFilter[ ] getChoosableFileFilters()`	It gets the list of user choosable file filters
`File getCurrentDirectory()`	It gets the current directory
`String getDescription (File f)`	It gets the file description

Table 14.19: Methods of the JFileChooser class

Method	Does this
`String getDialogTitle()`	It gets the string that goes in the file chooser's title bar
`int getDialogType()`	It gets the type of this dialog box
`FileFilter getFileFilter()`	It gets the currently selected file filter
`int getFileSelectionMode()`	It gets the current file-selection mode
`FileSystemView getFileSystemView()`	It gets the file system view
`FileView getFileView()`	It gets the current file view
`Icon getIcon(File f)`	It gets the icon for this file or type of file, depending on the system
`String getName(File f)`	It gets the file name
`File getSelectedFile()`	It gets the selected file
`File[ ] getSelectedFiles()`	It gets a list of selected files if the file chooser is set to allow multiple selections
`String getTypeDescription (File f)`	It gets the file type
`FileChooserUI getUI()`	It gets the UI object that implements the look and feel for this component
`String getUIClassID()`	It gets a string that specifies the name of the look and feel class that renders this component
`boolean isDirectorySelectionEnabled()`	It is a convenience method that determines whether directories are selectable
`Boolean isFileHidingEnabled()`	It returns True if hidden files are not shown in the file chooser
`boolean isFileSelectionEnabled()`	It is a convenience method that determines whether files are selectable based on the current file-selection mode
`boolean isMultiSelectionEnabled()`	It returns True if multiple files can be selected
`boolean isTraversable(File f)`	It returns True if the file (directory) can be visited
`protected String paramString()`	It gets a string representation of this `JFileChooser` object
`void removeActionListener (ActionListener l)`	It removes an action listener from the button
`boolean removeChoosableFileFilter (FileFilter f)`	It removes a filter from the list of choosable file filters
`void rescanCurrentDirectory()`	It rescans the file list from the current directory
`void resetChoosableFileFilters()`	It resets the choosable file filter list to its starting state
`void setAccessory (JComponent newAccessory)`	It sets the accessory component
`void setApproveButtonMnemonic (char mnemonic)`	It sets the approve button's mnemonic using a character
`void setApproveButtonMnemonic (int mnemonic)`	It sets the approve button's mnemonic using a numeric keycode
`void setApproveButtonText (String approveButtonText)`	It sets the text used in the approve button
`void setApproveButtonToolTipText (String toolTipText)`	It sets the tooltip text used in the approve button
`void setCurrentDirectory(File dir)`	It sets the current directory
`void setDialogTitle (String dialogTitle)`	It sets the string that goes in the file chooser window's title bar

Table 14.19: Methods of the JFileChooser class	
Method	**Does this**
`void setDialogType(int dialogType)`	It sets the type of this dialog box
`void setFileFilter(FileFilter filter)`	It sets the current file filter
`void setFileHidingEnabled (boolean b)`	It sets file hiding on or off
`void setFileSelectionMode(int mode)`	It sets the file chooser to allow the user to just select files, just select directories, or select both files and directories
`void setFileSystemView (FileSystemView fsv)`	It sets the file system view that the `JFileChooser` object uses
`void setFileView(FileView fileView)`	It sets the file view used to retrieve UI information, such as the icon that represents a file or the type description of a file
`void setMultiSelectionEnabled (boolean b)`	It sets the file chooser to allow multiple file selections
`void setSelectedFile(File file)`	It sets the selected file
`void setSelectedFiles (File[ ] selectedFiles)`	It sets the list of selected files if the file chooser is set to allow multiple selections
`protected void setup (FileSystemView view)`	It performs constructor initialization and setup
`int showDialog (Component parent, String approveButtonText)`	It displays a custom file chooser dialog box with a custom approve button
`int showOpenDialog (Component parent)`	It displays an "Open File" file chooser dialog box
`int showSaveDialog (Component parent)`	It displays a "Save File" file chooser dialog box
`void updateUI()`	It is called by the `UIFactory` class when the look and feel has changed

You can use the `JFileChooser` class's `showOpenDialog()` method to show a file chooser for finding files to open, and you can use the `showSaveDialog()` method to show a file chooser for specifying the file name and path to use to save a file. These methods return the following values:

- `APPROVE_OPTION`—It is returned if the user clicks an approve button, such as Save or Open.
- `CANCEL_OPTION`—It is returned if the user clicks Cancel.
- `ERROR_OPTION`—It is returned if there was an error.

You can get the selected file as a `File` object with the file chooser's `getSelectedFile()` method (you'll see the `File` class later in this book), and you can use the `File` class's `getPath()`, `getName()`, and other methods to return information about the file.

Let's look at an example that puts this to work. In this case, we display an `Open file` chooser when the user clicks a button (Figure 14.12). We let the user select a file, and then we display that file's name and path in a text field. Because you usually don't open files from applets (for security reasons), we'll make this example an application. Here's the code (note that we check the return value from `showOpenDialog()` to see whether the user clicked the Open button before displaying the file name in the text field):

```
import java.awt.*;
import java.io.File;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.filechooser.*;
public class DemoJFileChooser extends JFrame implements ActionListener {
         JFileChooser chooser = new JFileChooser();
         JButton button = new JButton("Display file chooser");
         JTextField text = new JTextField(30);
         public DemoJFileChooser() {
         super();
         Container c = getContentPane();
         c.setLayout(new FlowLayout());
```

```
            c.add(button);
            c.add(text);
            button.addActionListener(this); }
            public void actionPerformed(ActionEvent e) {
            int result = chooser.showOpenDialog(null);
            File file = chooser.getSelectedFile();
            if(result == JFileChooser.APPROVE_OPTION) {
            text.setText("You selected: " + file.getPath()); }
            else if(result == JFileChooser.CANCEL_OPTION) {
            text.setText("You clicked Cancel"); }
            }
            public static void main(String args[]) {
            JFrame f = new DemoJFileChooser();
            f.setBounds(200, 200, 400, 200);
            f.setVisible(true);
            f.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
            f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
            System.exit(0); }} );
            }
      }
```

The file chooser that this application displays appears in Figure 14.13. The user can browse and select a file (Figure 14.13). When he/she does, either by highlighting a file and clicking the Open button in the file chooser or by simply double-clicking the file in the file chooser, the file chooser closes, and the name and path of the selected file appear in the text field in the application, as you can see in Figure 14.14:



Figure 14.12: Display file chooser

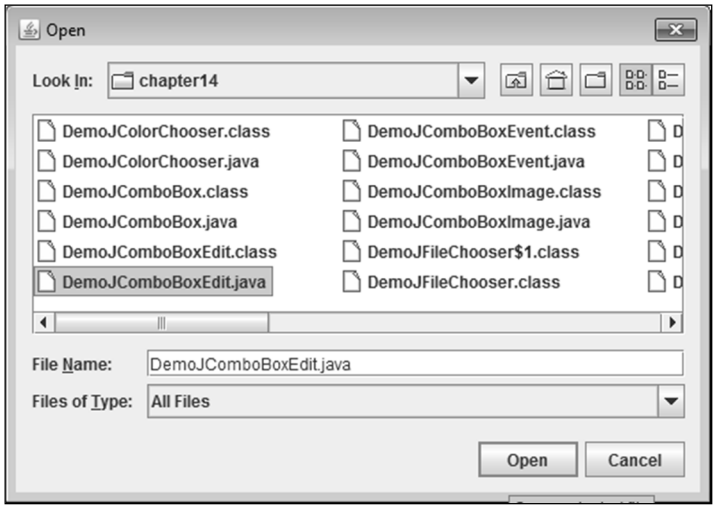


Figure 14.13: Displaying a file chooser

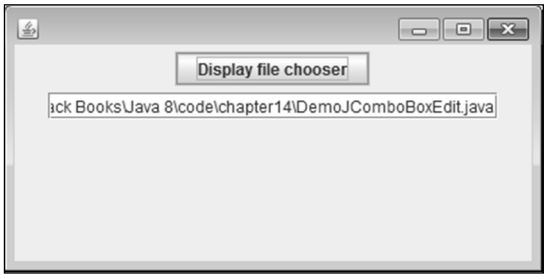


Figure 14.14: Determining the chosen file

That's all there is to it.

Creating File Chooser Filters

There are plenty of ways to customize file choosers, and one of them is to create file *filters*. You use filters to set the type of files—based on their extensions—that a file chooser will show. File filters are derived from the `FileFilter` class. To add a file filter to a file chooser, you use the `addChoosableFileFilter()` method of `JFileChooser`.

Let's look at an example in which we add two new file filters to a file chooser—one filter for `.gif` files and one for `.java` files. These two filters will be supported by the `MyFilter1` and `MyFilter2` classes, so we can use the `addChoosableFileFilter()` method to add them to a file chooser, like this:

```
import java.io.*;
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
         CODE = DemoJFileChooserFilter.class
         WIDTH = 200
         HEIGHT = 200 >
</APPLET>*/
public class DemoJFileChooserFilter extends JFrame implements ActionListener {
         JFileChooser filechooser = new JFileChooser();
         JButton button = new JButton("Display file chooser");
         JTextField text = new JTextField(20);
         public DemoJFileChooserFilter() {
  super();
  Container c = getContentPane();
  c.setLayout(new FlowLayout());
  c.add(button);
  c.add(text);
  button.addActionListener(this);
  filechooser.addChoosableFileFilter(new MyFilter1());
  filechooser.addChoosableFileFilter(new MyFilter2()); }
         public void actionPerformed(ActionEvent e) {
  int result = filechooser.showOpenDialog(null);
  if(result == JFileChooser.APPROVE_OPTION) {
  text.setText("You selected: " +
  filechooser.getSelectedFile().getPath()); }
         }
         public static void main(String a[]) {
  JFrame f = new DemoJFileChooserFilter();
  f.setBounds(200, 200, 300, 300);
  f.setVisible(true);
  f.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
  f.addWindowListener(new WindowAdapter() {
```

```
      public void windowClosing(WindowEvent e) {
      System.exit(0); }});
            }
   }
```

Now we have to create the `MyFilter1` and `MyFilter2` classes. These classes are derived from the abstract `FileFilter` class, and you'll find the methods of this class in Table 14.20:

Table 14.20: Methods of the FileFilter class

Method	Does this
`abstract boolean accept(File f)`	It returns True if the file is accepted by this filter
`abstract String getDescription()`	It returns the description of this filter

It's not difficult to implement the methods of the `FileFilter` class; the `accept()` method is passed a `File` object (you'll see the `File` class later in this book). It returns True if the file should be displayed (that is, if its extension is one that the filter accepts or, usually, if the file corresponds to a directory) and False otherwise. The `getDescription()` method returns a string that will be displayed in the file chooser to indicate to the user what types of files the filter is for.

The `MyFilter1` class will accept `.gif` files (described as "Gif Files (`*.gif`)" in this filter) and directories, and the `MyFilter2` class will filter `.java` files (described as "Java Files (`*.java`)" in this filter) and directories. Here's what these classes look like in code:

```
   import java.io.File;
   class MyFilter1 extends javax.swing.filechooser.FileFilter {
           public boolean accept(File file) {
           String extension = "";
           if(file.getPath().lastIndexOf('.') > 0)
           extension = file.getPath().substring(
     file.getPath().lastIndexOf('.') + 1).toLowerCase();
     if(extension != "") { return extension.equals("gif"); }
     else { return file.isDirectory(); }
     }
           public String getDescription() { return "Gif Files (*.gif)"; }
   }
   class MyFilter2 extends javax.swing.filechooser.FileFilter {
           public boolean accept(File file) {
     String extension = "";

     if(file.getPath().lastIndexOf('.') > 0)
     extension = file.getPath().substring(
     file.getPath().lastIndexOf('.') + 1).toLowerCase();
     if(extension != "") { return extension.equals("java"); }
     else { return file.isDirectory(); }
           }

           public String getDescription() { return "Java Files (*.java)"; }
   }
```

And that's it. The result of this code appears as shown in Figure 14.15, where you can see the `.java` filter in action:

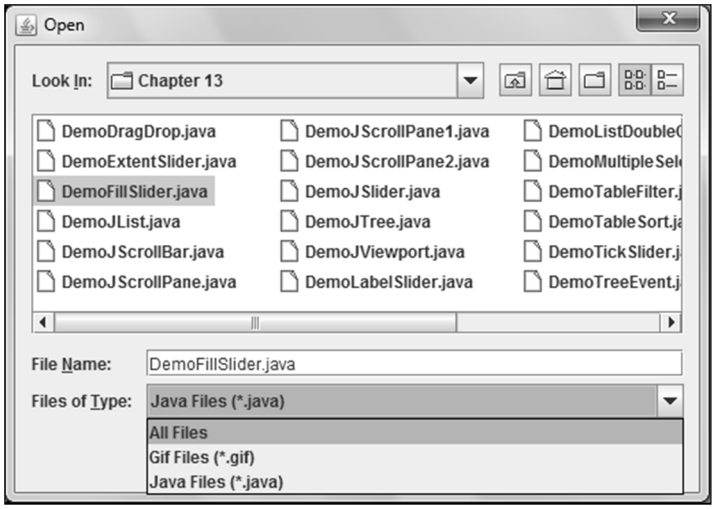


Figure 14.15: Using file filters in a file chooser

Using filters like this, you can make it easy for the user to search for files of a particular type.

Summary

In this chapter, you have learned about the Swings-based controls, combo boxes, progress bars, separators and classes for tooltips. In the next chapter, you'll learn about the panes and layouts part of Swing.



15

Swing—Layered Panes, Tabbed Panes, Split Panes, and Layouts

If you need an immediate solution to:	*See page:*
Understanding Swing Components and Z-order	523
Making Swing Components Transparent	524
Using Layered Panes	525
Creating Tabbed Panes	527
Specifying Tab Placement in the Tabbed Panes	532
Using Split Panes	535
Making Split Panes One-Touch Expandable	538
Setting Split Pane Orientation	539
Setting Split Pane Divider Size	540
Using the Box Layout Manager	541
Using the Box Class	543
Using the Overlay Layout Manager	547

In Depth

In this chapter, we'll take a look at some of the Swing lightweight containers: layered panes, tabbed panes, and split panes. Swing provides a number of lightweight containers, including `JPanel` (which you have already seen), to let programmers manage components in simple ways. Layered panes let you specify the *layer* of the components you add, making it easy to set the *Z-order* (that is, the stacking order in the Z axis, which points out of the screen) of components. Tabbed panes let you order the components much like adding them to a collection of tabbed file folders, and you can click the tabs to open each folder. Finally, split panes let you display two components side by side and adjust how much of each one is visible—a technique commonly used to support two views in the same model.

We'll also take a look at layouts in this chapter. You first saw the AWT layouts in Chapter 7. Swing supports all those layouts and two more: box layouts and overlay layouts. We'll consider these two new layouts in this chapter. Box layouts let you create rows or columns of components. Overlay layouts, as the name implies, let you draw overlapping components. Swing also defines the `Box` class, which lets you arrange components by using visual constructs, such as struts, rigid areas, and glue. You'll see all these in this chapter.

Layered Panes

Layered panes, themselves lightweight containers, are important panes of heavyweight Swing containers, such as `JApplet` and `JFrame`. They are divided into a number of vertical layers you can work with, and that Swing works with as well. This is one of the areas in which the implementation of Swing on top of AWT is not transparent to the programmer, because you can see the layers that Swing uses to display dialog boxes, drag components, pop-up menus, and so on. One of the most popular aspects of the layered pane is that it's home to the content pane.

Tabbed Panes

Dialog boxes that let users choose from many options have come to be organized by tabs as programs have become more complex and offer more options. Setup dialog boxes are customarily implemented this way, organizing program settings into tabs, such as Display, User Information, General, Files, and so on. Swing supports tabbed panes to let you create dialog boxes and programs of this kind. As you might expect, all the Swing capabilities are available here, such as displaying images in the tabs.

Split Panes

Another popular Graphical User Interface (GUI) programming technique today is to let the user *split* a view into a model's data, thus creating a new view into that data. Word processors often let users split their presentations into two panes, so they can move around in two areas of a document independently. Swing supports split panes to let programs present two components side by side. These components can represent views in the same or separate model data.

Layouts

You first saw the AWT layouts in Chapter 7, and Swing supports those layouts, as well as two more: box layouts and overlay layouts. Of the two, box layouts are the most popular because they let you arrange components into vertical and horizontal rows and columns, called *boxes*. In fact, Swing also supports the `Box` class, which goes a step further by letting you specify the spacing and arrangement of components, as you'll see in the chapter. The overlay layout manager lets you overlap components in a well-defined way, and although not as common as other layout managers, it has its uses.

That's it for the overview of what's in this chapter. As you can see, there's a lot coming up. It's time to turn to the "Immediate Solutions" section, starting with a look at stacking Swing components.

Immediate Solutions

Understanding Swing Components and Z-order

"Java's gone all wacky again," says the Novice Programmer. "I drew some components in the same area of my program by mistake, and they actually appeared on top of each other!" You smile and say, "That's perfectly possible. In fact, in Swing, it's not uncommon to overlap or overlay components." The NP says, "Weird!"

When you add lightweight components to a content pane, those components are simply drawn on that pane, so there's nothing to stop you from making those components overlap (that is, as long as you don't use a layout manager—except the overlay layout manager, which would stop overlaying from taking place). When components overlap, their z-order becomes important; *z-order* represents the relative placement of components along the z-axis, which goes out of the screen.

The lightweight components you add first will appear on top of those you add later. You can also add heavyweight AWT components to your program; and those components, which have their own operating system window, usually appear on top of Swing components.

Here's an example in which we remove the default border layout manager from a content pane and add a series of overlapping labels, showing that the first added label will appear on top of the rest. Here's what this looks like in code (note that we add a border to the labels to make the overlapping visible):

```
      import java.awt.*;
      import javax.swing.*;
      /* <APPLET
            CODE = DemoZOrder.class
            WIDTH = 250
            HEIGHT = 200 >
      </APPLET>*/
      public class DemoZOrder extends JApplet {
            JLabel labels[];
            public void init() {
            Container c = getContentPane();
            c.setLayout(null);
            labels = new JLabel[5];
            labels[0] = new JLabel("First");
            labels[0].setOpaque(true);
            labels[0].setBorder(BorderFactory.createEtchedBorder());
            c.add(labels[0]);
            labels[1] = new JLabel("Second");
            labels[1].setOpaque(true);
            labels[1].setBorder(BorderFactory.createEtchedBorder());
            c.add(labels[1]);
            labels[2] = new JLabel("Third");
            labels[2].setOpaque(true);
      labels[2].setBorder(BorderFactory.createEtchedBorder());
            c.add(labels[2]);
            labels[3] = new JLabel("Fourth");
            labels[3].setOpaque(true);
            labels[3].setBorder(BorderFactory.createEtchedBorder());
            c.add(labels[3]);
            labels[4] = new JLabel("Fifth");
      labels[4].setOpaque(true);
      labels[4].setBorder(BorderFactory.createEtchedBorder());
      c.add(labels[4]);
      for(int loop_index = 0; loop_index < 5; loop_index++) {
            labels[loop_index].setBounds(30 * loop_index, 30
            * loop_index, 110, 50); }
            }
   }
```

The result appears as shown in Figure 15.1:

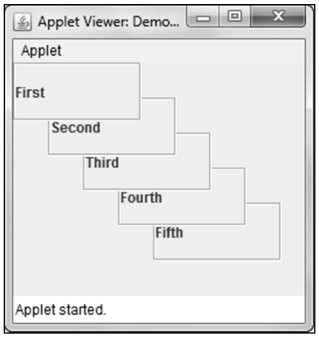


Figure 15.1: Overlapping controls

As you can see in the figure, the first label added appears on top of the others. The next added label appears on top of the one added after it and so on, thus defining a different Z-order for each label.

You might also notice that we used the `setOpaque()` method for each label to make sure that the labels underneath it don't show through—take a look at the next solution for more details.

Making Swing Components Transparent

"I just saw the strangest program;" the NP says, "you could actually see through controls to the ones underneath!" "That's not so strange," you say. "In fact, many Swing programs do that intentionally." "Tell me more!" says the NP.

Because lightweight controls are just drawn in their containers, you can make them *transparent*. When you make a control transparent, it doesn't draw its background when drawn or redrawn, so the controls underneath it can show through. This is a widely used technique that enables you to make it appear as though you are drawing components that are irregular and nonrectangular.

In the previous solution, we wrote an example in which we overlapped opaque Swing labels, but we can also explicitly make those labels transparent. You can set the transparency of Swing components with the `setOpaque()` method. Here's how we make the labels in the previous solution's example transparent (in fact, the calls to `setOpaque()` are actually unnecessary here because the default setting for labels is to make them transparent):

```
import java.awt.*;
import javax.swing.*;
/* <APPLET
      CODE = DemoTransparent.class
      WIDTH = 250
      HEIGHT = 200 >
</APPLET>*/
public class DemoTransparent extends JApplet {
      JLabel labels[];
      public void init() {
      Container c = getContentPane();
      c.setLayout(null);
      labels = new JLabel[5];
      labels[0] = new JLabel("First");
      labels[0].setOpaque(false);
      labels[0].setBorder(BorderFactory.createEtchedBorder());
      c.add(labels[0]);
      labels[1] = new JLabel("Second");
      labels[1].setOpaque(false);
      labels[1].setBorder(BorderFactory.createEtchedBorder());
      c.add(labels[1]);
      labels[2] = new JLabel("Third");
      labels[2].setOpaque(false);
```

```
      labels[2].setBorder(BorderFactory.createEtchedBorder());
            c.add(labels[2]);
            labels[3] = new JLabel("Fourth");
            labels[3].setOpaque(false);
            labels[3].setBorder(BorderFactory.createEtchedBorder());
            c.add(labels[3]);
            labels[4] = new JLabel("Fifth");
      labels[4].setOpaque(false);
      labels[4].setBorder(BorderFactory.createEtchedBorder());
      c.add(labels[4]);
      for(int loop_index = 0; loop_index < 5; loop_index++) {
            labels[loop_index].setBounds(30 * loop_index, 30
            * loop_index, 110, 50); }
            }
   }
```

The result appears in Figure 15.2, where you can see the overlapping transparent labels:

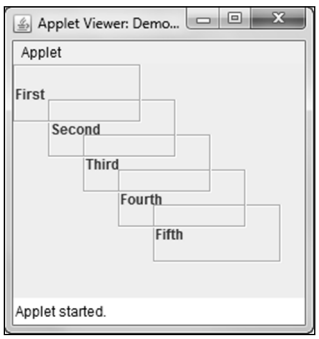


Figure 15.2: Making controls transparent

Note that there's an explicit way of handling overlapping controls without setting up your own layout manager. This involves a layered pane, which you can use in most programs. See the next solution for the details.

Using Layered Panes

"I'm having some trouble," the NP says. "I want to let the user drag components around in a Swing container; but when the components are dragged, they go *over* some other components and *under* others." You smile and say, "That's because you have to keep in mind that components are layered in Swing. You should put what you want to drag in the drag layer." "How's that?" the NP asks.

NOTE

You have already learned about the layered pane in Chapter 11.

You can use the `JLayeredPane` methods—`moveToFront()`, `moveToBack()`, and `setPosition()`—to reposition a component within its layer. The `setLayer()` method can also be used to change the component's current layer. We first covered `JLayeredPane` in Chapter 11 where in you could find the fields of the `JLayeredPane` class in Table 11.10, its constructors in Table 11.11, and its methods in Table 11.12.

Here's an example showing how to add components to a layered pane. In this case, we replace the content pane in an applet with a new-layered pane, and we add a label to each standard layer. To set the current layer in the layered pane, we use the `setLayer()` method; and to specify what layer to use, we use the constants defined in the `JLayeredPane` class, such as `JLayeredPane.PALETTE_LAYER`. One Swing curiosity is that constants, such as `JLayeredPane.PALETTE_LAYER` are actually `Integer` objects, not integers. However, methods, such as `setLayer()`, need integers, so you must use the `Integer` class's `intValue()` method to convert such constants into values you can use, like this:

```
JLayeredPane.PALETTE_LAYER.intValue()
```

Here's how we add labels to all the layers outlined in the preceding list:

```
import java.awt.*;
import javax.swing.*;
/* <APPLET
         CODE = DemoLayeredPane.class
         WIDTH = 250
         HEIGHT = 200 >
</APPLET>*/
public class DemoLayeredPane extends JApplet {
         JLayeredPane layeredpane = new JLayeredPane();
         JLabel labels[];
         public void init() {
         setContentPane(layeredpane);
         layeredpane.setLayout(null);
         labels = new JLabel[6];
  labels[0] = new JLabel("Content layer");
  labels[0].setBorder(BorderFactory.createEtchedBorder());
  layeredpane.setLayer(labels[0],
  JLayeredPane.FRAME_CONTENT_LAYER.intValue());
  layeredpane.add(labels[0]);
  labels[1] = new JLabel("Default layer");
  labels[1].setBorder(BorderFactory.createEtchedBorder());
  layeredpane.setLayer(labels[1],
  JLayeredPane.DEFAULT_LAYER.intValue());
  layeredpane.add(labels[1]);
  labels[2] = new JLabel("Palette layer");
  labels[2].setBorder(BorderFactory.createEtchedBorder());
  layeredpane.setLayer(labels[2],
  JLayeredPane.PALETTE_LAYER.intValue());
  layeredpane.add(labels[2]);
  labels[3] = new JLabel("Modal layer");
  labels[3].setBorder(BorderFactory.createEtchedBorder());
  layeredpane.setLayer(labels[3],
  JLayeredPane.MODAL_LAYER.intValue());
  layeredpane.add(labels[3]);
  labels[4] = new JLabel("Popup layer");
  labels[4].setBorder(BorderFactory.createEtchedBorder());
  layeredpane.setLayer(labels[4],
  JLayeredPane.POPUP_LAYER.intValue());
  layeredpane.add(labels[4]);
  labels[5] = new JLabel("Drag layer");
  labels[5].setBorder(BorderFactory.createEtchedBorder());
  layeredpane.setLayer(labels[5],
  JLayeredPane.DRAG_LAYER.intValue());
  layeredpane.add(labels[5]);
  for(int loop_index = 0; loop_index < 5; loop_index++) {
         labels[loop_index].setBounds(30 * loop_index, 30
         * loop_index, 110, 50); }
}
}
```

The result of this code appears as shown in Figure 15.3, where you can see what layer is on top of what other layer:

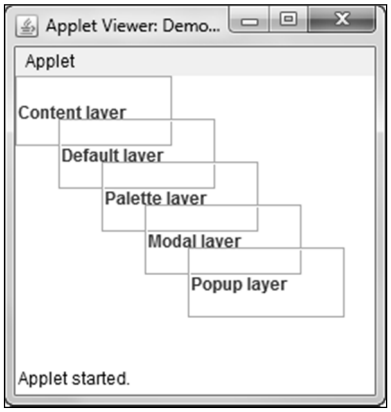


Figure 15.3: Adding components to a layered pane

Now we have worked with all the standard layers in a layered pane (you can also define your own layers numerically).

In Figure 15.3, note that the drag layer appears on top, which makes sense because you want components the user drags to ride above other components. To implement dragging by using the drag layer, you add the component you want to drag to the drag layer, make sure the layout manager has been set to null, and then use the component's `setLocation()` method to position the component in response to the user's mouse actions.

Creating Tabbed Panes

"My program has so many settings now that the setup dialog box is a little larger than the screen itself," says the NP. "How much larger?" you ask. "About four times," the NP says. "Ah," you say, "it sounds like it's time to arrange those options by using a tabbed pane."

The Tabbed panes are useful when a number of components are to be arranged in less space. They provide the users a content view based upon tabbed files and folders. When a user clicks on a tab, it seems that the "folder" corresponding to it opens and displays its contents. The `JTabbedPane` class is responsible for providing the Tabbed pane support in Swing. Its inheritance diagram is as follows:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.JTabbedPane
```

You'll find the fields for the `JTabbedPane` class in Table 15.1, its constructors in Table 15.2, and its methods in Table 15.3.

Table 15.1: Fields of the JTabbedPane class

Field	Does this
`protected ChangeEvent changeEvent`	The `ChangeEvent` object
`protected ChangeListener changeListener`	The `ChangeListener` object added to the model
`protected SingleSelectionModel model`	The default selection model
`static int SCROLL_TAB_LAYOUT`	It's a tab layout policy for providing a subset of available tabs when all the tabs will not fit within a single run
`protected int tabPlacement`	Where the tabs are placed
`static int WRAP_TAB_LAYOUT`	The tab layout policy for wrapping tabs in multiple runs when all tabs will not fit within a single run

Table 15.2: Constructors of the JTabbedPane class	
Constructor	**Does this**
`JTabbedPane()`	It constructs an empty `TabbedPane` object
`JTabbedPane(int tabPlacement)`	It constructs an empty `TabbedPane` object with the indicated tab placement of TOP, BOTTOM, LEFT, or RIGHT
`JTabbedPane(int tabPlacement, int tabLayoutPolicy)`	It creates an empty `TabbedPane` with the specified tab placement and tab layout policy

Table 15.3: Methods of the JTabbedPane class	
Method	**Does this**
`Component add(Component component)`	It adds a component
`Component add(Component component, int index)`	It adds a component at the indicated tab index
`void add(Component component, Object constraints)`	It adds a component to the tabbed pane
`void add(Component component, Object constraints, int index)`	It adds a component at the indicated tab index
`Component add(String title, Component component)`	It adds a component with the indicated tab title
`void addChangeListener(ChangeListener l)`	It adds a `ChangeListener` object to this tabbed pane
`void addTab(String title, Component component)`	It adds a tab represented by a title and no icon
`void addTab(String title, Icon icon, Component component)`	It adds a tab represented by a title and/or icon, either of which can be null
`void addTab(String title, Icon icon, Component component, String tip)`	It adds a tab and tip represented by a title and/or icon, either of which can be null
`protected ChangeListener createChangeListener()`	It overrides this method to return a subclass of `ModelListener` or another `ChangeListener` implementation
`protected void fireStateChanged()`	It sends a `ChangeEvent` object to each listener
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` object associated with this `JComponent` object
`Color getBackgroundAt(int index)`	It gets the tab background color at index
`Rectangle getBoundsAt(int index)`	It gets the tab bounds at index
`Component getComponentAt(int index)`	It gets the component at index
`Icon getDisabledIconAt(int index)`	It gets the tab disabled icon at index
`Color getForegroundAt(int index)`	It gets the tab foreground color at index
`Icon getIconAt(int index)`	It gets the tab icon at index
`SingleSelectionModel getModel()`	It gets the model associated with this tabbed pane
`Component getSelectedComponent()`	It gets the currently selected component for this tabbed pane
`int getSelectedIndex()`	It gets the currently selected index for this tabbed pane
`int getTabCount()`	It gets the number of tabs in this tabbed pane
`Component getTabComponentAt(int index)`	It returns the tab component at index
`int getTabPlacement()`	It gets the placement of the tabs for this tabbed pane
`int getTabRunCount()`	It gets the number of tab runs currently used to display the tabs
`String getTitleAt(int index)`	It gets the tab title at index

Table 15.3: Methods of the JTabbedPane class

Method	Does this
`String getToolTipText(MouseEvent event)`	It gets the tooltip text for the component determined by the mouse event location
`String getToolTipTextAt(int index)`	It returns the tab tooltip text at index
`TabbedPaneUI getUI()`	It gets the UI object that implements the look and feel for this component
`String getUIClassID()`	It gets the name of the UI class that implements the look and feel for this component
`int indexOfComponent(Component component)`	It gets the index of the tab for the indicated component
`int indexOfTab(Icon icon)`	It gets the first tab index with a given icon
`int indexOfTab(String title)`	It gets the first tab index with a given title and gets -1 if no tab has this title
`int indexOfTabComponent(Component tabComponent)`	It returns the index of the tab for the specified tab component
`void insertTab(String title, Icon icon, Component component, String tip, int index)`	It inserts a component represented by a title and icon
`boolean isEnabledAt(int index)`	It gets whether the tab at index is currently enabled
`protected String paramString()`	It gets a string representation of this `JTabbedPane` object
`void remove(Component component)`	It removes the tab that corresponds to the indicated component
`void remove(int index)`	It removes the tab and component that correspond to the specified index
`void removeAll()`	It removes all the tabs from the tabbed pane
`void removeChangeListener(ChangeListener l)`	It removes a `ChangeListener` object from this tabbed pane
`void removeTabAt(int index)`	It removes the tab at index
`void setBackgroundAt(int index, Color background)`	It sets the background color
`void setComponentAt(int index, Component component)`	It sets the component at `index` to component
`void setDisabledIconAt(int index, Icon disabledIcon)`	It sets the disabled icon at `index` to `icon`, which can be null
`void setDisplayedMnemonicIndexAt(int tabIndex, int mnemonicIndex)`	It provides a hint to the look and feel as to which character in the text should be decorated to represent the mnemonic
`void setEnabledAt(int index, boolean enabled)`	It sets whether the tab at `index` is enabled
`void setForegroundAt(int index, Color foreground)`	It sets the foreground color
`void setIconAt(int index, Icon icon)`	It sets the icon at `index` to `icon`, which can be null
`void setMnemonicAt(int tabIndex, int mnemonic)`	It sets the keyboard mnemonic for accessing the specified tab
`void setModel(SingleSelectionModel model)`	It sets the model to be used with this tabbed pane
`void setSelectedComponent(Component c)`	It sets the selected component for this tabbed pane
`void setSelectedIndex(int index)`	It sets the selected index for this tabbed pane
`void setTabComponentAt(int index, component component)`	It sets the component that is responsible for rendering the title for the specified tab

Table 15.3: Methods of the JTabbedPane class	
Method	**Does this**
`void setTabLayoutPolicy(int tabLayoutPolicy)`	It sets the policy which the tabbed pane will use in laying out the tabs when all the tabs will not fit within a single run
`void setTabPlacement(int tabPlacement)`	It sets the tab placement for this tabbed pane
`void setTitleAt(int index, String title)`	It sets the title
`void setUI(TabbedPaneUI ui)`	It sets the UI object that implements the look and feel for this component
`void updateUI()`	It is called by `UIManager` when the look and feel has changed

An example will make all this information clearer. In this case, we create a tabbed pane with three tabs and give each tab its own `JPanel` object with a label in it. The user will be able to move from tab to tab, seeing all the labels in turn. We start by creating the panels we'll use:

```
import java.awt.*;
  import javax.swing.*;
  /*<APPLET
        CODE = DemoTabbedPane.class
        WIDTH = 500
        HEIGHT = 200 >
</APPLET>*/
public class DemoTabbedPane extends JApplet {
  public void init() {
Container c = getContentPane();
JTabbedPane tabbedpane = new JTabbedPane();
JPanel jp1 = new JPanel();
JPanel jp2 = new JPanel();
JPanel jp3 = new JPanel();
jp1.add(new JLabel("First Panel"));
jp2.add(new JLabel("Second Panel"));
jp3.add(new JLabel("Third Panel"));
. . .
```

Now we are ready to add these panels to a tabbed pane, so we create a new tabbed pane and use the `addTab()` method to add three tabs to it. We pass the caption to `addTab()` method for each tab, the image icon to use in each tab, the components to add to each tab (which are usually objects of the `JPanel` class), and a tooltip for each tab, like this:

```
tabbedpane.addTab("Tab 1",new ImageIcon("tabb.jpg"),jp1, "First Tab");
tabbedpane.addTab("Tab 2",new ImageIcon("tabb.jpg"),jp2, "Second Tab");
tabbedpane.addTab("Tab 3",new ImageIcon("tabb.jpg"),jp3, "Third Tab");
c.setLayout(new BorderLayout());
c.add(tabbedpane); }
}
```

The result appears as shown in Figure 15.4:

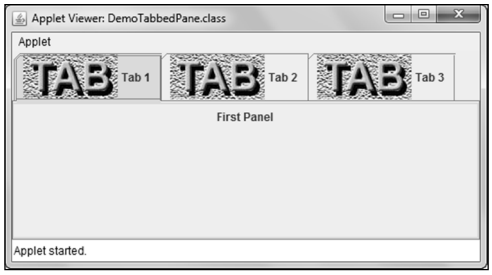


Figure 15.4: Creating a tabbed pane

As you can see in the figure, each tab has its own image and caption, as specified to the `addTab()` method. Clicking a new tab opens a new "folder" with a new label in it. That's all there is to it.

You can do more with tabbed panes—specify the placement you want for the tabs. Here is another example that will make this information clearer. In this case, we create a tabbed pane with 12 tabs, for months of the year, and give each tab its own `JPanel` object with a label in it. The user will be able to move the alignment of tab and the way tab should be displayed on the frame, seeing all the labels in turn.

```
import javax.swing.*;
import java.text.*;
import java.awt.*;
public class DemoTabsLayout {
  public static void main(String args[]) {
        String[] month = new DateFormatSymbols().getShortMonths();
              JTabbedPane tabbedpane = new JTabbedPane();
        tabbedpane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_LAYOUT);
        for (int i=0; i<12; i++) {
              JPanel tabpanel = new JPanel(new BorderLayout());
              JButton tabbutton = new JButton(month[i]);
              tabpanel.add(tabbutton);
              tabbedpane.add(month[i], tabpanel); }
        JFrame frm = new JFrame("Tabs Layout");
        frm.getContentPane().add(tabbedpane, BorderLayout.CENTER);
        frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frm.setSize(300, 200);
        frm.setVisible(true); }
}
```

The result appears as shown in Figure 15.5:

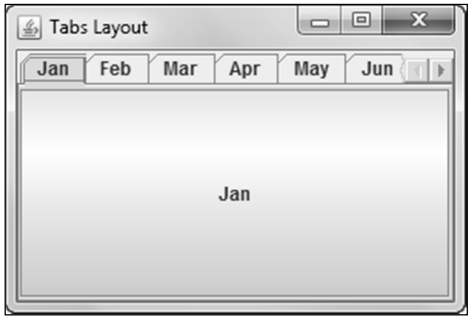


Figure 15.5: Setting the tab policy

You can see the change if you add the following line of code to the preceding code (after the method `setTabLayoutPolicy()`).

```
tabbedpane.setTabPlacement(JTabbedPane.LEFT);
```

This will bring the tab to the left side of the screen, as shown in Figure 15.6:

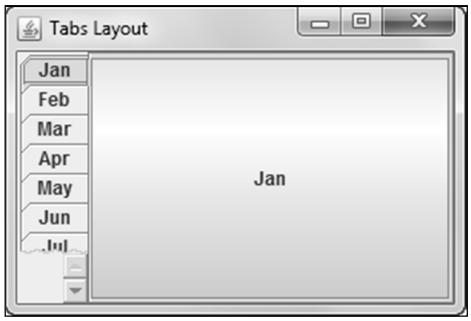


Figure 15.6: Setting the tab on left of window

Now we'll make some changes to the previously mentioned code, say we delete the following code statement:

```
tabbedpane.setTabPlacement(JTabbedPane.LEFT);
```

and modify the code statement:

```
tabbedpane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_LAYOUT);
```

to:

```
tabbedpane.setTabLayoutPolicy(JTabbedPane.WRAP_TAB_LAYOUT);
```

The result of this modification appears in Figure 15.7:

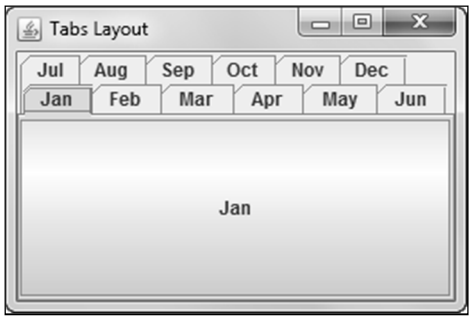


Figure 15.7: Setting the Tab Strip Layout

Specifying Tab Placement in the Tabbed Panes

The Big Boss is looking over your shoulder and says, "No, no, that's all wrong— the tabs should go on the *left* side." "You keep changing your mind," you say. "Are you sure you want the tabs on the left?" "No, no," says the BB, "on the *right* side, like I said."

You can specify where the tabs appear in a tabbed pane—top, bottom, left, or right—using the `setTabPlacement()` method. Here's an example showing how that works. In this case, we let the user select the orientation of the tabs in a tabbed pane—left, right, top, or bottom—by clicking a button. Here's the code:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

/*
<APPLET
  CODE = DemoTabbedPaneOrient.class
  WIDTH = 400
  HEIGHT = 200 >
</APPLET>
*/

public class DemoTabbedPaneOrient extends JApplet implements
ActionListener {
         JTabbedPane tabbedpane = new JTabbedPane(SwingConstants.BOTTOM);
         JButton button1, button2, button3, button4;
         public void init() {
  Container c = getContentPane();
  JPanel buttonPanel = new JPanel();
  JPanel jp1 = new JPanel();
  JPanel jp2 = new JPanel();
  JPanel jp3 = new JPanel();
  tabbedpane.setTabPlacement(JTabbedPane.TOP);
  tabbedpane.addTab("Panel 1",
  new ImageIcon("tabb.jpg"),
  jp1, "First Panel");
  tabbedpane.addTab("Panel 2",
  new ImageIcon("tabb.jpg"),
  jp2, "Second Panel");
```

```
      tabbedpane.addTab("Panel 3",
      new ImageIcon("tabb.jpg"),
      jp3, "Third Panel");
      button1 = new JButton("Top");
      button2 = new JButton("Left");
      button3 = new JButton("Right");
      button4 = new JButton("Bottom");
      buttonPanel.add(button1);
      buttonPanel.add(button2);
      buttonPanel.add(button3);
      buttonPanel.add(button4);
      button1.addActionListener(this);
      button2.addActionListener(this);
      button3.addActionListener(this);
      button4.addActionListener(this);
      c.setLayout(new BorderLayout());
      c.add(tabbedpane, BorderLayout.CENTER);
      c.add(buttonPanel, BorderLayout.SOUTH);
            }
      public void actionPerformed(ActionEvent e) {
            if(e.getSource() == button1) {
            tabbedpane.setTabPlacement(JTabbedPane.TOP); }
            else if(e.getSource() == button2) {
            tabbedpane.setTabPlacement(JTabbedPane.LEFT); }
            else if(e.getSource() == button3) {
            tabbedpane.setTabPlacement(JTabbedPane.RIGHT); }
            else if(e.getSource() == button4) {
            tabbedpane.setTabPlacement(JTabbedPane.BOTTOM); }
            tabbedpane.validate(); }
   }
```

The result of this code appears as shown in Figure 15.8:

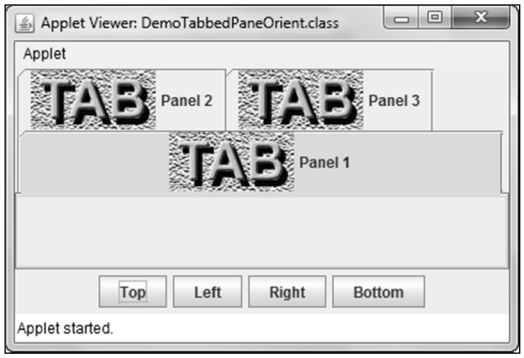


Figure 15.8: Setting a tabbed pane's orientation

When the user clicks a button, the tabbed pane responds by displaying the tabs in the corresponding orientation, thus allowing the users to customize the tabbed pane as they like. The Big Boss would be proud.

Now we'll make another example that will enable users to choose from the choices given under each tab. Here's the code:

```
   import javax.swing.*;
   /* <APPLET code="PickupMenu"
   WIDTH=400
   HEIGHT=100>
   </APPLET>
   */
   public class PickupMenu extends JApplet {
     public void init() {
           try {
                   SwingUtilities.invokeAndWait(new Runnable()
                   {
```

```
                        public void run() { createGUI(); }});
                }
        catch (Exception e) {
        System.out.println("Sorry, your order cannot be processed " + e); }
        }
   private void createGUI() {
        JTabbedPane tabbedpane = new JTabbedPane();
        tabbedpane.addTab("Pizza", new Pizza());
        tabbedpane.addTab("Toppings", new Toppings());
        tabbedpane.addTab("Beverage", new Beverage());
        tabbedpane.addTab("Add-ons", new AdOn());
        add(tabbedpane); }
}

class Pizza extends JPanel {
   public Pizza() {
        JRadioButton radio1 = new JRadioButton("Classic");
        add(radio1);
        JRadioButton radio2 = new JRadioButton("Hand-tossed");
        add(radio2);
        JRadioButton radio3 = new JRadioButton("Special Crust");
        add(radio3); }
}
class Toppings extends JPanel {
   public Toppings() {
        JCheckBox check1 = new JCheckBox("Onion");
        add(check1);
        JCheckBox check2 = new JCheckBox("Capsicum");
        add(check2);
        JCheckBox check3 = new JCheckBox("Tomato");
        add(check3);
        JCheckBox check4 = new JCheckBox("Plain Cheese");
        add(check4); }
}
class Beverage extends JPanel {
   public Beverage() {
        JComboBox combobox = new JComboBox();
        combobox.addItem("Coke");
        combobox.addItem("Pepsi");
        combobox.addItem("7'up");
        combobox.addItem("Sprite");
        add(combobox); }
}
class AdOn extends JPanel {
   public AdOn() {
        JButton button1 = new JButton("Garlic Bread");
        add(button1);
        JButton button2 = new JButton("Chocolate Cake");
        add(button2);
        JButton button3 = new JButton("Salads");
        add(button3); }
}
```

The result of this code appears in Figure 15.9:

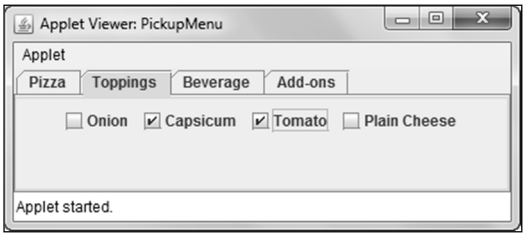


Figure 15.9: Selecting options within a tab

Using Split Panes

The Product Support Specialist is back and unhappy. "About your new word processor program," the PSS says, "users are complaining. When they edit a document more than a thousand pages long and have to refer to another part of the document, it can take a long time to scroll back and forth." "Who's editing documents more than a thousand pages long?" you want to know. "How about adding a split pane to the word processor?" the PSS asks.

You can use split panes to display two components side by side, which can include two views in the same model. Using split panes, the user can manage two components, dragging the divider between the components to show more or less of each, as needed. Split panes are supported by the `JSplitPane` class, and here's the inheritance diagram for that class:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.JSplitPane
```

You'll find the fields of the `JSplitPane` class in Table 15.4, its constructors in Table 15.5, and its methods in Table 15.6:

Table 15.4: Fields of the JSplitPane class

Field	Does this
`static String BOTTOM`	It is used to add a component at the bottom
`static String CONTINUOUS_LAYOUT_PROPERTY`	The bound property name for `continuousLayout`
`protected boolean continuousLayout`	It returns True if views are continuously redisplayed while resizing
`static String DIVIDER`	It is used to add a component that will represent the divider
`static String DIVIDER_SIZE_PROPERTY`	The bound property name for a border
`protected int dividerSize`	The size of the divider
`static int HORIZONTAL_SPLIT`	A horizontal split indicates the components are split along the x axis
`static String LAST_DIVIDER_LOCATION_PROPERTY`	The bound property for `lastLocation`
`protected int lastDividerLocation`	The previous location of the split pane.
`static String LEFT`	It is used to add a component to the left of the other component
`protected Component leftComponent`	The left or top component
`static String ONE_TOUCH_EXPANDABLE_PROPERTY`	The bound property for `oneTouchExpandable`
`protected boolean oneTouchExpandable`	It returns True if the split pane is one-touch expandable
`protected int orientation`	It indicates how the views are split
`static String ORIENTATION_PROPERTY`	The bound property name for the orientation (horizontal or vertical)
`static String RIGHT`	It is used to add a component to the right of the other component
`protected Component rightComponent`	The right or bottom component
`static String TOP`	It is used to add a component above the other component
`static int VERTICAL_SPLIT`	A vertical split indicates the components are split along the y-axis
`static String DIVIDER_LOCATION_PROPERTY`	It is used to bound property for the `dividerLocation`

Table 15.5: Constructors of the JSplitPane class	
Constructor	**Does this**
`JSplitPane()`	It constructs a new `JSplitPane` object with two buttons for the components
`JSplitPane(int newOrientation)`	It constructs a new `JSplitPane` object configured with the indicated orientation and no continuous layout
`JSplitPane(int newOrientation, boolean newContinuousLayout)`	It constructs a new `JSplitPane` object with the indicated orientation and redrawing style
`JSplitPane(int newOrientation, boolean newContinuousLayout, Component newLeftComponent, Component newRightComponent)`	It constructs a new `JSplitPane` object with the indicated orientation and redrawing style and with the indicated components
`JSplitPane(int newOrientation, Component newLeftComponent, Component newRightComponent)`	It constructs a new `JSplitPane` object with the indicated orientation and with the specified components, which does not perform continuous redrawing

Table 15.6: Methods of the JSplitPane class	
Method	**Does this**
`protected void addImpl(Component comp, Object constraints, int index)`	It adds `comp` with the given constraints
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` object
`Component getBottomComponent()`	It gets the component below or to the right of the divider
`int getDividerLocation()`	It gets the location of the divider from the look and feel implementation
`int getDividerSize()`	It gets the size of the divider
`int getLastDividerLocation()`	It gets the last location of the divider
`Component getLeftComponent()`	It gets the component to above or left of the divider
`int getMaximumDividerLocation()`	It gets the maximum location of the divider from the look and feel implementation
`int getMinimumDividerLocation()`	It gets the minimum location of the divider from the look and feel implementation
`int getOrientation()`	It gets the orientation
`double getResizeWeight()`	It returns the number that determines how extra space is distributed
`Component getRightComponent()`	It gets the component to the right of or below the divider
`Component getTopComponent()`	It gets the component above or to the left of the divider
`SplitPaneUI getUI()`	It gets the `SplitPaneUI` object that's providing the current look and feel
`String getUIClassID()`	It gets the name of the look and feel class that renders this component
`boolean isContinuousLayout()`	It returns True if the child components are continuously redisplayed and laid out during user intervention
`boolean isOneTouchExpandable()`	It returns True if the pane provides a UI widget to collapse/expand the divider
`boolean isValidateRoot()`	It returns True if the root validates
`protected void paintChildren(Graphics g)`	It paints child components
`protected String paramString()`	It gets a string representation of this `JSplitPane` object
`void remove(Component component)`	It removes the child component from the pane
`void remove(int index)`	It removes the component at the indicated index
`void removeAll()`	It removes all the child components from the receiver

Table 15.6: Methods of the JSplitPane class

Method	Does this
`void resetToPreferredSizes()`	It repositions the `JSplitPane` object based on the preferred size of the children
`void setBottomComponent(Component comp)`	It sets the component below or to the right of the divider
`void setContinuousLayout(boolean newContinuousLayout)`	It determines whether the child components are continuously redisplayed during user actions
`void setDividerLocation(double proportionalLocation)`	It sets the divider location as a percentage of the `JSplitPane` object's size
`void setDividerLocation(int location)`	It sets the location of the divider
`void setDividerSize(int newSize)`	It sets the size of the divider
`void setLastDividerLocation(int newLastLocation)`	It sets the last location of the divider to `newLastLocation`
`void setLeftComponent(Component comp)`	It sets the component to the left or above the divider
`void setOneTouchExpandable(boolean newValue)`	It determines whether the `JSplitPane` object provides a UI widget on the divider for quickly expanding/ collapsing the divider
`void setOrientation(int orientation)`	It sets the orientation or how the splitter is divided
`void setResizeWeight(double value)`	It specifies how to distribute extra space when the size of the split pane changes
`void setRightComponent(Component comp)`	It sets the component to the right of or below the divider
`void setTopComponent(Component comp)`	It sets the component above or to the left of the divider
`void setUI(SplitPaneUI ui)`	It sets the look and feel object that draws this component
`void updateUI()`	It is called by the `UIManager` object when the look and feel has changed

We'll put the `JSplitPane` class to use in an example here. In this case, we add two text fields to a split pane. That's easy enough to do—we just create two text fields and a split pane, like this:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
CODE = DemoSplitPane.class
WIDTH = 400
HEIGHT = 200 >
</APPLET>*/
public class DemoSplitPane extends JApplet implements ActionListener {
JTextField textfield1 = new JTextField("TextField 1");
JTextField textfield2 = new JTextField("TextField 2");
JSplitPane splitpane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
textfield1, textfield2);
. . .
```

TIP

If you don't explicitly add components to a split pane, it will display two buttons.

Now we just add the new split pane to the applet's content pane as follows:

```
public class DemoSplitPane extends JApplet implements ActionListener {
   JTextField textfield1 = new JTextField("TextField 1");
   JTextField textfield2 = new JTextField("TextField 2");
   JSplitPane splitpane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
   textfield1, textfield2); public void init() {
```

```
            Container c = getContentPane();
            c.add(splitpane, BorderLayout.CENTER); }
        public void actionPerformed(ActionEvent e) { }
        }
```

That's all it takes. The result of this code appears as shown in Figure 15.10:

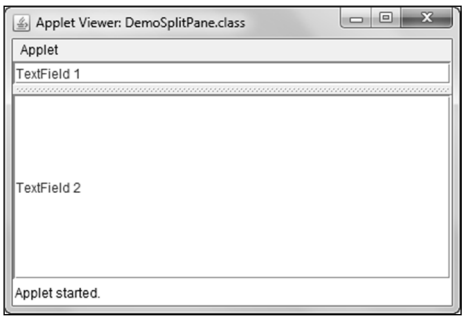


Figure 15.10: Creating a split pane

From now on, the user can adjust the divider between the text fields, showing more or less of each, as needed. The split pane will manage the view of each text field from now on.

NOTE

You will learn how to use creating scroll panes in Chapter 13 of this book.

Making Split Panes One-Touch Expandable

"Take a look at the new split pane I have put in my program," the NP says. "That's a good one," you agree, "but why isn't it one-touch expandable?" The NP replies, "Huh?"

Making a split pane one-touch expandable adds small arrow buttons to its divider. When an arrow is clicked, it expands one of the panes to take up the full split pane. This is a useful mechanism if you want to maximize a component in a split pane, because split panes don't allow components to be reduced below a certain minimum.

You can make a split pane one-touch expandable with the split pane `setOneTouchExpandable()` method. To show you how this works, we'll add a button to the split pane example in the previous solution that, when clicked, will make the split pane one-touch expandable. To add that button, we first create a new panel and add it to the layout of the applet, like this:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
CODE = DemoExpandableSplitPane.class
WIDTH = 400
HEIGHT = 200 >
</APPLET>*/
public class DemoExpandableSplitPane extends JApplet implements ActionListener {
JButton button;
JTextField textfield1 = new JTextField("TextField 1");
JTextField textfield2 = new JTextField("TextField 2");
JSplitPane splitpane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
textfield1, textfield2);
public void init() {
Container c = getContentPane();
JPanel panel = new JPanel();
button = new JButton("Make one-touch expandable");
button.addActionListener(this);
panel.add(button);
c.add(splitpane, BorderLayout.CENTER);
c.add(panel, BorderLayout.SOUTH); }
```

```
            public void actionPerformed(ActionEvent e) {
            if(e.getSource() == button) {
      splitpane.setOneTouchExpandable(true); }
            }
   }
```

The result of this code appears as shown in Figure 15.11:

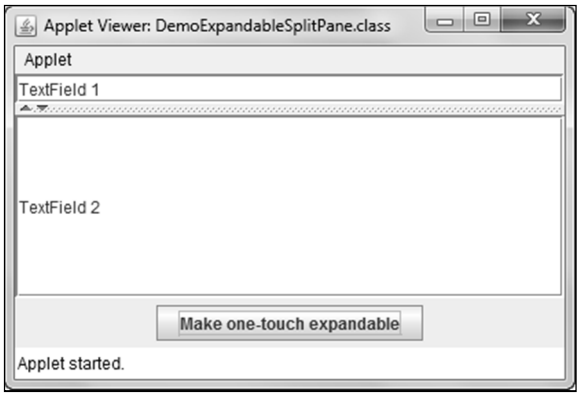


Figure 15.11: Making a split pane one-touch expandable

When the user clicks the button, two arrows appear in the split pane's divider, as you can see in the figure. Now the split pane is one-touch expandable; for example, when the user clicks the up arrow, the bottom text field expands to fill the split pane (except for the divider, which is still visible, so the user can click the down arrow to restore the original appearance).

Setting Split Pane Orientation

"No, no, no," says the Big Boss impatiently, "we don't want any vertical splits in split panes. We want *horizontal* splits." "OK," you say. "I can do that. When do you want it by the way?" "Is it done yet?" the BB asks.

You can set the orientation of the divider in a split pane by passing the `setOrientation()` method the constant `JSplitPane.HORIZONTAL_SPLIT` or `JSplitPane.VERTICAL_SPLIT`. We'll add this capability to the split pane example from the previous solution with a new button that lets the user specify a horizontal split, like this:

```
    import java.awt.*;
    import javax.swing.*;
    import java.awt.event.*;
    /* <APPLET
    CODE = DemoSplitPaneOrient.class
    WIDTH = 400
    HEIGHT = 200 >
    </APPLET>*/
    public class DemoSplitPaneOrient extends JApplet implements ActionListener {
    JButton button1, button2;
    JTextField textfield1 = new JTextField("TextField 1");
    JTextField textfield2 = new JTextField("TextField 2");
    JSplitPane splitpane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
    textfield1, textfield2);
    public void init() {
    Container c = getContentPane();
    JPanel panel = new JPanel();
    button1 = new JButton("Make one-touch expandable");
    button1.addActionListener(this);
    panel.add(button1);
    button2 = new JButton("Make split horizontal");
    button2.addActionListener(this);
    panel.add(button2);
```

```
    c.add(splitpane, BorderLayout.CENTER);
    c.add(panel, BorderLayout.SOUTH); }
            public void actionPerformed(ActionEvent e) {
      if(e.getSource() == button1) {
      splitpane.setOneTouchExpandable(true); }
      if(e.getSource() == button2) {
      splitpane.setOrientation(JSplitPane.HORIZONTAL_SPLIT); }
            }
    }
```

The result of this code appears as shown in Figure 15.12:

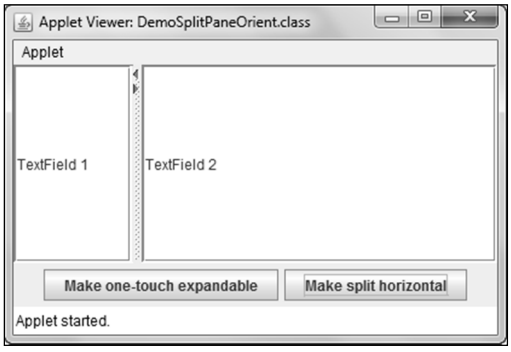


Figure 15.12: Setting split pane orientation

You can see the new horizontally oriented split in the figure.

Setting Split Pane Divider Size

You can set the size of the divider that appears in a split pane using the `setDividerSize()` method, passing it a new divider width in pixels. Here's an example in which we add a new button to the example from the previous solution. When the user clicks this button, we get the current divider width with the `getDividerSize()` method, add 5 pixels to it, and use the result as the new divider width. Here's what the code looks like:

```
    import java.awt.*;
    import javax.swing.*;
    import java.awt.event.*;
    /*<APPLET
    CODE = DemoSplitPaneDivider.class
    WIDTH = 600
    HEIGHT = 200 >
    </APPLET>*/
    public class DemoSplitPaneDivider extends JApplet implements ActionListener {
    JButton button1, button2, button3;
    JTextField textfield1 = new JTextField("TextField 1");
    JTextField textfield2 = new JTextField("TextField 2");
    JSplitPane splitpane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
    textfield1, textfield2);
    public void init() {
    Container c = getContentPane();
    JPanel panel = new JPanel();
    button1 = new JButton("Make one-touch expandable");
    button1.addActionListener(this);
    panel.add(button1);
    button2 = new JButton("Make split horizontal");
    button2.addActionListener(this);
    panel.add(button2);
    button3 = new JButton("Increase divider size");
    button3.addActionListener(this);
    panel.add(button3);
    c.add(splitpane, BorderLayout.CENTER);
```

```
    c.add(panel, BorderLayout.SOUTH); }
            public void actionPerformed(ActionEvent e) {
      if(e.getSource() == button1) {
      splitpane.setOneTouchExpandable(true); }
      if(e.getSource() == button2) {
      splitpane.setOrientation(JSplitPane.HORIZONTAL_SPLIT); }
      if(e.getSource() == button3) {
      splitpane.setDividerSize(splitpane.getDividerSize() + 5);
      }
            }
    }
```

The result appears as shown in Figure 15.13:

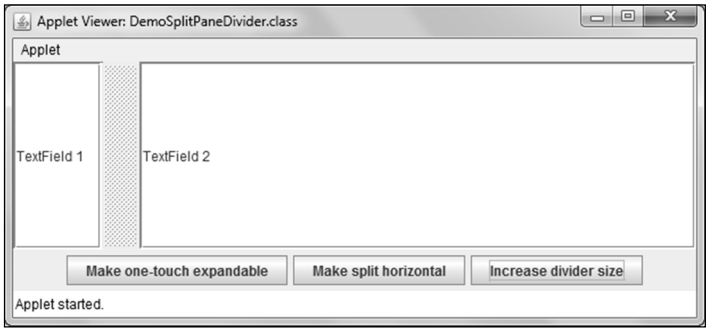


Figure 15.13: Setting a split pane divider's width

When the user clicks the `Increase divider size` button, the divider increases in size by 5 pixels each time.

Using the Box Layout Manager

"I have got a lot of drawing tools in my program," the NP says, "and I want to arrange them vertically in one large column. Any suggestions?" "Sure," you say, "use a box layout." "OK," says the NP, "how?"

Swing introduces two new layouts: the box layout and the overlay layout. We'll first take a look at the box layout in this solution and the overlay layout later. You can use box layouts to lay components out along the x-axis in a row or along the y-axis in a column. Here's the inheritance diagram for the `BoxLayout` class, which supports the box layout:

```
    java.lang.Object
      |____javax.swing.BoxLayout
```

You'll find the fields of the `BoxLayout` class in Table 15.7, its constructor in Table 15.8, and its methods in Table 15.9.

Table 15.7: Fields of the BoxLayout class

Field	Does this
`static int X_AXIS`	It indicates that the components should be arranged left to right
`static int Y_AXIS`	It indicates that the components should be arranged top to bottom
`static int LINE_AXIS`	It indicates that the components should be laid out in the direction of a line of text as determined by the target container's `ComponentOrientation` property
`static int PAGE_AXIS`	It indicates that the components should be laid out in the direction that lines flow across a page as determined by the target container's `ComponentOrientation` property

Table 15.8: The constructor of the BoxLayout class

Constructor	Does this
`BoxLayout(Container target, int axis)`	It constructs a box layout manager

Table 15.9: Methods of the BoxLayout class	
Method	**Does this**
`float getLayoutAlignmentX(Container target)`	It gets the alignment along the x axis for the container
`float getLayoutAlignmentY(Container target)`	It gets the alignment along the y axis for the container
`void invalidateLayout(Container target)`	It indicates that a child has changed its layout information
`void layoutContainer(Container target)`	It is called by the AWT when the indicated container needs to be laid out
`Dimension maximumLayoutSize(Container target)`	It gets the maximum dimensions the target container can use to lay out the components it contains
`Dimension minimumLayoutSize(Container target)`	It gets the minimum dimensions needed to lay out the components contained in the indicated target container
`Dimension preferredLayoutSize(Container target)`	It gets the preferred dimensions for this layout, given the components in the indicated target container

Here's an example in which we arrange three panels using box layouts—two vertically and one horizontally—and add four text fields to each panel. This simply entails using a new box layout manager in each panel like this:

```
import java.awt.*;
import javax.swing.*;
/* <APPLET
CODE = DemoBoxLayout.class
WIDTH = 400
HEIGHT = 200 >
</APPLET>*/
public class DemoBoxLayout extends JApplet {
public void init() {
Container c = getContentPane();
JPanel panel1, panel2, panel3;
panel1 = new JPanel();
panel2 = new JPanel();
panel3 = new JPanel();
panel1.setLayout(new BoxLayout(panel1, BoxLayout.Y_AXIS));
panel2.setLayout(new BoxLayout(panel2, BoxLayout.X_AXIS));
panel3.setLayout(new BoxLayout(panel3, BoxLayout.Y_AXIS));
c.setLayout(new FlowLayout());
panel1.add(new JTextField("TextField 1"));
panel1.add(new JTextField("TextField 2"));
panel1.add(new JTextField("TextField 3"));
panel1.add(new JTextField("TextField 4"));
panel2.add(new JTextField("TextField 1"));
panel2.add(new JTextField("TextField 2"));
panel2.add(new JTextField("TextField 3"));
panel2.add(new JTextField("TextField 4"));
panel3.add(new JTextField("TextField 1"));
panel3.add(new JTextField("TextField 2"));
panel3.add(new JTextField("TextField 3"));
panel3.add(new JTextField("TextField 4"));
c.add(panel1);
c.add(panel2);
c.add(panel3); }
}
```

The result appears in Figure 15.14, where you can see the three box layouts:

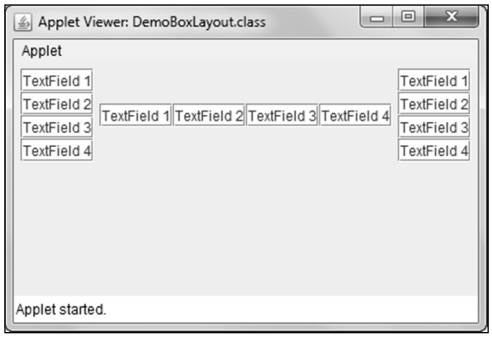


Figure 15.14: Creating box layouts

This example was relatively easy to create and run.

However, there's more box layout power available—take a look at the `Box` class in the next solution.

Using the **Box** Class

"Box layouts are fine," says the NP, "but I need more control. For example, I want to space the controls in a row myself, but I can't do that with a box layout." "Sure you can," you say, "if you use the `Box` class." "How's that?" the NP asks.

The `Box` class uses a box layout and adds methods to let you work with the following constructs to extend that layout:

- **Glue**—It expands to fill the empty region between components. You can use glue to space components evenly.
- **Struts**—The vertical or horizontal distances that you specify and add to the layout to space components as you want them. Struts are fixed in one dimension and fill the other dimension as needed.
- **Rigid areas**—Rectangular areas that do not resize themselves. Here's the inheritance diagram for the `Box` class:

```
java.lang.Object
   |____java.awt.Component
         |____java.awt.Container
                |____javax.swing.JComponent
                       |____javax.swing.Box
```

You'll find the fields of the `Box` class in Table 15.10, its constructor in Table 15.11, and its methods in Table 15.12.

Table 15.10: The fields of the Box class

Field	Does this
`protected AccessibleContext accessibleContext`	The `AccessibleContext` object
`protected EventListenerList listenerList`	The list of event listeners for this component
`public static final string TOOL_TIP_TEXT_KEY`	The comment to display when the cursor is over the component
`protected transient ComponentUI ui`	The look and feel delegate for this component
`public static final int UNDEFINED_cONDITION`	The constant used by some of the APIs to mean that no condition is defined
`public static final int WHEN_ANCESTOR_OF_FOCUSED_COMPONENT`	The constant used for `registerKeyboardAction` that means the command should be invoked when the receiving component is an ancestor of the focused component or is itself the focused component

Table 15.10: The fields of the Box class	
Field	**Does this**
`public static final int WHEN_FOCUSED`	The constant used for `registerKeyboardAction` that means the command should be invoked when the component has the focus
`public static final int WHEN_IN_FOCUSED_WINDOW`	The constant used for `registerKeyboardAction` that means the command should be invoked when the receiving component is in the window that has the focus or is itself the focused component
`public static final float BOTTOM_ALIGNMENT`	The specification of an alignment to the bottom of the component
`public static final float CENTER_ALIGNMENT`	The specification of an alignment to the center of the component
`public static final float LEFT_ALIGNMENT`	The specification of an alignment to the left side of the component
`public static final float RIGHT_ALIGNMENT`	The specification of an alignment to the right side of the component
`public static final float TOP ALIGNMENT`	The specification of an alignment to the top of the component
`static final int ABORT`	The image being tracked asynchronously is aborted before the production completes
`static final int ALLBITS`	The image previously drawn is complete and can be drawn again in its final form
`static final int ERROR`	The image being tracked asynchronously encounters an error and no further information is displayed
`static final int FRAMEBITS`	An another complete frame of a multi- frame image previously drawn becomes available
`static final int HEIGHT`	The height of the base image becomes available
`static final int PROPERTIES`	The properties of an image becomes available
`static final int SOMEBITS`	The more number of pixels required to draw a scaled variation of the image becomes available
`static final int WIDTH`	The width of the base image becomes available

Table 15.11: The constructor of the Box class	
Constructor	**Does this**
`Box(int axis)`	It constructs a box that displays its components along its specified axis

Table 15.12: The methods of the Box class	
Method	**Does this**
`static Component createGlue()`	It constructs a glue component
`public static Box createVerticalBox()`	It constructs a box that displays its components from top to bottom
`static Component createHorizontalGlue()`	It constructs a horizontal glue component
`static Component createHorizontalStrut(int width)`	It constructs an invisible, fixed-width strut
`static Component createRigidArea(Dimension d)`	It constructs an invisible component that's always the indicated size
`public static Component createRigidArea(Dimension d)`	It constructs a box that displays its components from top to bottom
`static Component createVerticalGlue()`	It constructs a vertical glue component

Table 15.12: The methods of the Box class	
Method	**Does this**
`static Component createVerticalStrut(int height)`	It constructs an invisible, fixed-height strut
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` object
`void setLayout(LayoutManager l)`	It throws an `AWTError` object (a `Box` object can only use `BoxLayout`)
`protected void paintComponent(Graphic g)`	It is used for painting the box

Here's an example using the `Box` class. In this case, we use the `createGlue()`, `createRigidArea()`, `createHorizontalStrut()`, and `createVerticalStrut()` methods of the `Box` class to create glue, rigid areas, and struts. These components can be added to box layouts, so we'll create six panels, each with a box layout, and add text areas to the panels using glue, struts, and rigid areas to show how all this works. Here's the code:

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
/*<APPLET
CODE = DemoBox.class
WIDTH = 550
HEIGHT = 350 >
</APPLET>*/
public class DemoBox extends JApplet {
public void init() {
  Container c = getContentPane();
  c.setLayout(new FlowLayout());
  JPanel panel1 = new JPanel();
  panel1.setBorder(
BorderFactory.createTitledBorder(       BorderFactory.createEtchedBorder(), "Glue"));
  panel1.setLayout(new BoxLayout(panel1, BoxLayout.X_AXIS));
  panel1.add(Box.createGlue());
  panel1.add(new JTextField("TextField 1"));
  panel1.add(Box.createGlue());
  panel1.add(new JTextField("TextField 2"));
  panel1.add(Box.createGlue());
  panel1.add(new JTextField("TextField 3"));
  panel1.add(Box.createGlue());
  c.add(panel1);
  JPanel panel2 = new JPanel();
 panel2.setBorder(
        BorderFactory.createTitledBorder(
        BorderFactory.createEtchedBorder(), "Struts"));
        panel2.setLayout(new BoxLayout(panel2, BoxLayout.X_AXIS));
        panel2.add(new JTextField("TextField 1"));
        panel2.add(Box.createHorizontalStrut(20));
        panel2.add(new JTextField("TextField 2"));
        panel2.add(Box.createHorizontalStrut(20));
        panel2.add(new JTextField("TextField 3"));
        c.add(panel2);
        JPanel panel3 = new JPanel();
        panel3.setBorder(BorderFactory.createTitledBorder
        (BorderFactory.createEtchedBorder(), "Rigid"));
        panel3.setLayout(new BoxLayout(panel3, BoxLayout.X_AXIS));
        panel3.add(Box.createRigidArea(new Dimension(10, 40)));
        panel3.add(new JTextField("TextField 1"));
        panel3.add(Box.createRigidArea(new Dimension(10, 40)));
        panel3.add(new JTextField("TextField 2"));
        panel3.add(Box.createRigidArea(new Dimension(10, 40)));
        panel3.add(new JTextField("TextField 3"));
```

```
            c.add(panel3);
            JPanel panel4 = new JPanel();
            panel4.setBorder(BorderFactory.createTitledBorder
            (BorderFactory.createEtchedBorder(),"Glue"));
            panel4.setLayout(new BoxLayout(panel4, BoxLayout.Y_AXIS));
            panel4.add(Box.createGlue());
            panel4.add(new JTextField("TextField 1"));
            panel4.add(Box.createGlue());
            panel4.add(new JTextField("TextField 2"));
            panel4.add(Box.createGlue());
            panel4.add(new JTextField("TextField 3"));
            panel4.add(Box.createGlue());
            c.add(panel4);
            JPanel panel5 = new JPanel();
            panel5.setBorder(BorderFactory.createTitledBorder
            (BorderFactory.createEtchedBorder(),    "Struts"));
            panel5.setLayout(new BoxLayout(panel5, BoxLayout.Y_AXIS));
            panel5.add(new JTextField("TextField 1"));
            panel5.add(Box.createVerticalStrut(30));
            panel5.add(new JTextField("TextField 2"));
            panel5.add(Box.createVerticalStrut(30));
            panel5.add(new JTextField("TextField 3"));
            c.add(panel5);
            JPanel panel6 = new JPanel();
            panel6.setBorder(BorderFactory.createTitledBorder
            (BorderFactory.createEtchedBorder(),    "Rigid"));
            panel6.setLayout(new BoxLayout(panel6, BoxLayout.Y_AXIS));
            panel6.add(Box.createRigidArea(new Dimension(40, 60)));
            panel6.add(new JTextField("TextField 1"));
            panel6.add(Box.createRigidArea(new Dimension(40, 60)));
            panel6.add(new JTextField("TextField 2"));
            panel6.add(Box.createRigidArea(new Dimension(40, 60)));
            panel6.add(new JTextField("TextField 3"));
            c.add(panel6); }
    }
```

The result of this code appears as shown in Figure 15.15:

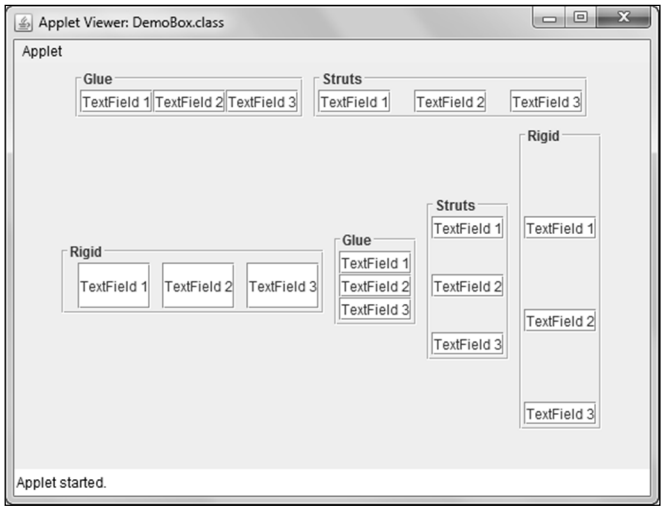


Figure 15.15: Using the Box class

As you can see, using the `Box` class is an easy way to extend box layouts, allowing you to customize the spacing between the components.

Using the Overlay Layout Manager

"My program is getting pretty cramped," says the NP. "Is there any easy way I can overlap the controls?" "Are you sure you want to do that?" you ask. "Sure," says the NP. "Well," you say, "you can remove the layout manager and locate your controls yourself, or you can use the overlay manager." "There's an overlay manager?" the NP asks. "Tell me more!"

You can use the overlay manager to overlay components in Swing. This layout manager is supported by the `OverlayLayout` class. Here's the inheritance diagram for this class:

```
java.lang.Object
  |____javax.swing.OverlayLayout
```

You'll find the constructor for this class in Table 15.13 and its methods in Table 15.14:

Table 15.13: The constructor of the OverlayLayout class

Constructor	Does this
`OverlayLayout(Container target)`	It constructs an overlay layout manager

Table 15.14: The methods of the OverlayLayout class

Method	Does this
`void addLayoutComponent(Component comp, Object constraints)`	It adds the indicated component to the layout
`void addLayoutComponent(String name, Component comp)`	It adds the indicated component to the layout
`float getLayoutAlignmentX(Container target)`	It gets the alignment along the x axis for the container
`float getLayoutAlignmentY(Container target)`	It gets the alignment along the y axis for the container
`void invalidateLayout(Container target)`	It indicates a child has changed its layout related information, which causes any cached calculations to be flushed
`void layoutContainer(Container target)`	It is called by the AWT when the indicated container needs to be laid out
`Dimension maximumLayoutSize(Container target)`	It gets the maximum dimensions needed to layout the components
`Dimension minimumLayoutSize(Container target)`	It gets the minimum dimensions needed to layout the components
`Dimension preferredLayoutSize(Container target)`	It gets the preferred dimensions for this layout, given the components
`void removeLayoutComponent(Component comp)`	It removes the indicated component from the layout
`Container getTarget()`	It returns the container that uses this layout manager

Here's how you use an overlay layout. In this kind of a layout, you add components to a container so that their alignment points are at the same location. Each of these components also has an alignment attribute between 0.0 and 1.0, which you set with the `setAlignmentX()` and `setAlignmentY()` methods (the default value is 0.5). These specify where the alignment point is in each dimension. For example, an x alignment attribute of 0.5 refers to the center of the component in the x direction.

After you set the minimum, maximum, and preferred size of the components, the overlay layout manager attempts to resize the components, so their alignment points overlap, and it maintains that overlap, if possible, even when the container is resized.

Let's look at an example to make this clearer. In this case, we overlap two text fields, specify their minimum, maximum, and preferred sizes, and set their alignment points, so one component will appear in the upper left and the other in the lower right. Here's the code:

```
import java.awt.*;
import javax.swing.*;
```

```
/*<APPLET
CODE = DemoOverlayLayout.class
WIDTH = 300
HEIGHT = 300 >
</APPLET>*/
public class DemoOverlayLayout extends JApplet {
public void init() {
   Container c = getContentPane();
   JPanel panel = new JPanel();
   panel.setLayout(new OverlayLayout(panel));
   panel.setBackground(Color.white);
   panel.setBorder(
         BorderFactory.createTitledBorder(
         BorderFactory.createEtchedBorder(),
         "Overlay Layout"));
   JTextField textfield1 = new JTextField("TextField 1");
   JTextField textfield2 = new JTextField("TextField 2");
   textfield1.setMinimumSize(new Dimension(35, 35));
   textfield2.setMinimumSize(new Dimension(35, 35));
   textfield1.setPreferredSize(new Dimension(120, 120));
   textfield2.setPreferredSize(new Dimension(120, 120));
   textfield1.setMaximumSize(new Dimension(140, 140));
   textfield2.setMaximumSize(new Dimension(140, 140));
   textfield1.setAlignmentX(0.3f);
   textfield1.setAlignmentY(0.3f);
   textfield2.setAlignmentX(0.9f);
   textfield2.setAlignmentY(0.9f);
   panel.add(textfield1);
   panel.add(textfield2);
   c.setLayout(new FlowLayout());
   c.add(panel); }
}
```

The result appears as shown in Figure 15.16, where you can see the two overlapped text fields:

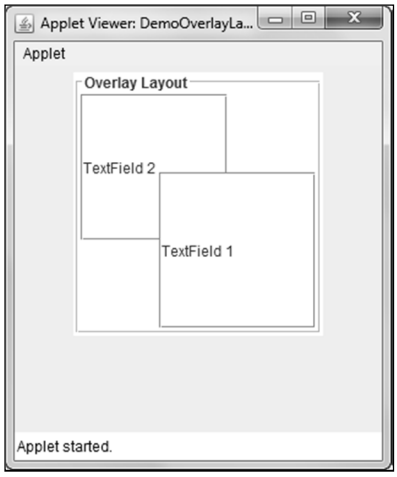


Figure 15.16: Using the overlay layout manager

Summary

In this chapter, you have learned about different panes such as layered, tabbed, and split. You have also come across how to make split panes one-touch expandable, set split pane orientation, use of box layout manager, and the Box class. Lastly, we discussed use of overlay layout manager.

In the next chapter, you'll learn about the menus and toolbars put a dash instead an important part of any application.



16

Swing—Menus and Toolbars

If you need an immediate solution to:	*See page:*
Creating a Menu Bar	551
Creating a Menu	552
Creating a Menu Item	555
Creating a Basic Menu System	557
Adding Images to Menu Items	559
Creating Checkbox Menu Items	560
Creating Radio Button Menu Items	562
Creating Submenus	564
Creating Menu Accelerators	565
Enabling/Disabling and Changing Menu Items at Runtime	567
Adding and Removing Menu Items at Runtime	568
Adding Buttons and Other Controls to Menus	569
Creating Pop-Up Menus	570
Creating Toolbars	574
Adding Combo Boxes and Other Controls to Toolbars	576

In Depth

In this chapter, we'll cover two of the most important Swing components—menus and toolbars. Both of these components are familiar to all GUI users, and you'll see how to work with them in Swing. Let's see first what Swing has to offer here in the overview itself.

Menus

Swing menus provide some substantial improvements over AWT menus, such as the ability to display images, the ability to set a menu item's look and feel, and, notably, the ability to display menus in applet windows. Swing menus are supported by the `JMenuBar`, `JMenu`, and `JMenuItem`classes, which support menu bars, menus, and menu items, respectively. As in AWT, Swing menus are actually built by using buttons behind the scenes, so you can use action listeners with them.

In this chapter, we'll take a look at what Swing menus have to offer, including creating basic menus, adding images to menu items, creating checkbox and radio button menu items and submenus, adding controls such as buttons to menus, creating menu accelerators and mnemonics, and more. We'll also take a look at creating pop-up menus here. Swing menus are more complex and powerful than their AWT components, as you'll see.

Toolbars

Toolbars are popular GUI controls, and they are new in Swing. Like menus, toolbars are built by the use of buttons in Swing. They provide a bar of buttons that are clickable, just like menu items. In fact, toolbars and menus are closely related— you'll often add buttons to toolbars that represent popular menu items to save the user from the trouble of opening a menu.

We'll know about Swing toolbars, which are supported by the `JtoolBar` class, including adding images to toolbar buttons, letting the user align the toolbar against any edge of a window, and adding other controls, such as combo boxes, to toolbars.

That's it for the overview of what's in this chapter. As you can see, there's a lot coming up. It's time to turn to the "Immediate Solutions" section, which starts with a look at stacking Swing components.

Immediate Solutions

Creating a Menu Bar

The Novice Programmer appears and says, "The Big Boss says I need to add a menu system to my program. How in the world do I do that?" "With three components:" you say, "menu bars, menus, and menu items. Get some coffee and we'll start with menu bars." "OK," the NP says.

To add a menu system to a Swing program, you must first create a menu bar using the `JmenuBar` class. You can add the menu bar to applets or frame windows with the `setJMenuBar()` method. Here's the inheritance diagram for the `JmenuBar` class:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.JMenuBar
```

You can find the constructor for `JmenuBar` in Table 16.1 and its methods in Table 16.2:

Table 16.1: The constructor of the JMenuBar class

Constructor	Does this
`JMenuBar()`	It constructs a new menu bar

Table 16.2: Methods of the JMenuBar class

Method	Does this
`JMenu add(JMenu c)`	It appends the indicated menu to the end of the menu bar
`void addNotify()`	It overrides `JComponent.addNotify` to register this menu bar
`AccessibleContextgetAccessibleContext()`	It gets the `AccessibleContext` object associated with its `JMenuBar`
`Component getComponent()`	It is implemented to be a `MenuElement` object
`Component getComponent(int i)`	It gets the component at the indicated index
`intgetComponentIndex(Componentc)`	It gets the index of the indicated component
`JMenugetHelpMenu()`	It gets the help menu for the menu bar
`Insets getMargin()`	It gets the margin between the menu bar's border and its menus
`JMenugetMenu(int index)`	It gets the menu at the indicated position in the menu bar
`intgetMenuCount()`	It gets the number of items in the menu bar
`SingleSelectionModelgetSelectionModel()`	It gets the model object that handles single selections
`MenuElement[] getSubElements()`	It returns the menus in its menu bar
`MenuBarUIgetUI()`	It gets the menu bar's current UI
`String getUIClassID()`	It gets the name of the `look and feel` class that renders this component
`booleanisBorderPainted()`	It returns True if the menu bar's border should be painted
`booleanisSelected()`	It returns True if the menu bar has a component selected
`void menuSelectionChanged(booleanisIncluded)`	It is implemented to be a `MenuElement` object but does nothing currently
`protected void paintBorder (Graphics g)`	It paints the menu bar's border
`protected String paramString()`	It gets a string representation of this `JMenuBar`

Table 16.2: Methods of the JMenuBar class	
Method	**Does this**
`protected boolean processKeyBinding(KeyStrokeks, KeyEvent e, int condition, boolean pressed)`	They are subclassed to check all the child menus
`void processKeyEvent(KeyEvent e, MenuElement[] path, MenuSelectionManager manager)`	It is implemented to be a `MenuElement` object but does nothing
`void processMouseEvent(MouseEvent event, MenuElement[] path, MenuSelectionManager manager)`	It is implemented to be a `MenuElement` object but does nothing
`void removeNotify()`	It overrides `JComponent.removeNotify` to unregister this menu bar
`void setBorderPainted (boolean b)`	It sets whether the border should be painted
`void setHelpMenu(JMenu menu)`	It sets the help menu that appears when the user selects the Help option in the menu bar
`void setMargin(Insets m)`	It sets the margin between the menu bar's border and its menus
`void setSelected(Component sel)`	It sets the selected component
`void setSelectionModel (SingleSelectionModelmodel)`	It sets the model object to handle single selections
`void setUI(MenuBarUIui)`	It sets the `look and feel` object that renders this component
`void updateUI()`	It is called by `UIFactory` when the look and feel has changed

We'll put the `JmenuBar` class to work upon the next few solutions while creating a basic menu system. The next step is to create menus to add to the menu bar, and we'll do that in the next solution.

Creating a Menu

The NP is back and says, "I have created a menu bar, but nothing appears in it—what's up?" You smile and say, "You have got to add menus to it explicitly." "Ah," says the NP, "now you can show me how that works."

You create the menus in a menu bar with the `Jmenu` class. Here's the inheritance diagram for that class (note that `Jmenu` is actually subclassed from `JMenuItem`, which is the case because `JMenuItem`, which is subclassed from `AbstractButton`, can respond to button clicks):

```
java.lang.Object
   |____java.awt.Component
         |____java.awt.Container
               |____javax.swing.JComponent
                     |____javax.swing.AbstractButton
                           |____javax.swing.JMenuItem
                                 |____javax.swing.JMenu
```

You'll find the field of the `JMenu`class in Table 16.3, its constructors in Table 16.4, and its methods in Table 16.5:

Table 16.3: The field of the JMenu class	
Field	**Does this**
`protected JMenu.WinListenerpopupListener`	It is the pop-up's window-closing listener

Table 16.4: Constructors of the JMenu class	
Constructor	**Does this**
`JMenu()`	It constructs a new `JMenu`
`JMenu(Action a)`	It constructs a menu whose properties are taken from the Action supplied

Table 16.4: Constructors of the JMenu class	
Constructor	**Does this**
`JMenu(String s)`	It constructs a new `JMenu` with the string as its text
`JMenu(String s, boolean b)`	It constructs a new `JMenu` with the string as its text and specifies whether it has a tear-off menu, as indicated by the boolean value

Table 16.5: Methods of the JMenu class	
Method	**Does this**
`JMenuItem add(Action a)`	It constructs a new menu item attached to the indicated `Action` object and appends it to the end of this menu
`Component add(Component c)`	It appends a component to the end of this menu
`Component add(Component c, int index)`	It adds the specified component to this container at the given position
`JMenuItem add(JMenuItemmenuItem)`	It appends a menu item to the end of this menu
`JMenuItem add(String s)`	It constructs a new menu item with the indicated text and appends it to the end of this menu
`void addMenuListener (MenuListener l)`	It adds a listener for menu events
`void addSeparator()`	It appends a new separator to the end of the menu
`void applyComponentOrientation (ComponentOrientation o)`	It sets the `ComponentOrientation` property of this menu and all components contained within it
`protected PropertyChangeListener createActionChangeListener (JMenuItem b)`	It creates an action change listener
`protected JmenuItem createActionComponent(Action a)`	It is a factory method that creates the `JMenuITem` object for actions added to the `JMenu`
`protected JMenu.WinListener createWinListener(JPopupMenu p)`	It creates a window-closing listener
`void doClick(intpressTime)`	It programmatically performs a click action
`protected void fireMenuCanceled()`	It notifies all listeners that have registered for this notification that this event occurred
`protected void fireMenuDeselected()`	It notifies all listeners that have registered for this notification
`protected void fireMenuSelected()`	It notifies all listeners that have registered for this notification
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` object
`Component getComponent()`	It returns the `java.awt.Component`object used to paint this `MenuElement`
`int getDelay()`	It gets the suggested delay before the menu's `PopupMenu` is popped up or down
`JMenuItem getItem(intpos)`	It gets the `JMenuItem` object at the indicated position
`int getItemCount()`	It gets the number of items on the menu, including separators
`Component getMenuComponent(int n)`	It gets the component at position `n`
`int getMenuComponentCount()`	It gets the number of components on the menu
`Component[] getMenuComponents()`	It gets an array of the menu's subcomponents
`MenuListener[] getMenuListeners()`	It gets an array of all the `MenuListeners` added to this `JMenu` with `addMenuListener`()
`JpopupMenu getPopupMenu()`	It gets the pop-up menu associated with this menu
`protected Point getPopupMenuOrigin()`	It computes the origin for the `JMenu's` popup menu

Table 16.5: Methods of the JMenu class

Method	Does this
`MenuElement[] getSubElements()`	It gets an array containing the submenu components for this menu component
`String getUIClassID()`	It gets the name of the `look and feel` class that renders this component
`JMenuItem insert (Action a, intpos)`	It inserts a new menu item attached to the indicated `Action` object at a given position
`JMenuItem insert (JMenuItem mi, intpos)`	It inserts the indicated `JMenuItem` object at a given position
`void insert(String s, intpos)`	It inserts a new menu item with the indicated text at a given position
`void insertSeparator (int index)`	It inserts a separator at the indicated position
`boolean isMenuComponent (Component c)`	It returns True if the indicated component exists in the submenu hierarchy
`boolean isPopupMenuVisible()`	It returns True if the menu's pop-up window is visible
`boolean isSelected()`	It returns True if the menu is currently selected (popped up)
`boolean isTearOff()`	It returns True if the menu can be torn off
`boolean isTopLevelMenu()`	It returns True if the menu is a top-level menu
`void menuSelectionChanged (booleanisIncluded)`	It is called when the menu bar selection changes
`protected String paramString()`	It gets a string representation of this `JMenu`
`protected void processKeyEvent(KeyEvent e)`	It overrides `processKeyEvent` to process events
`void remove(Component c)`	It removes the component
`void remove(intpos)`	It removes the menu item at the given position
`void remove(JMenuItem item)`	It removes the indicated menu item
`void removeAll()`	It removes all menu items from this menu
`void removeMenuListener (MenuListener l)`	It removes a listener for menu events
`void setAccelerator (KeyStrokekeyStroke)`	It is not defined for `JMenu`
`void setComponentOrientation (ComponentOrientation o)`	It sets the language-sensitive orientation that is to be used to order the elements or text within this component
`void setDelay(int d)`	It sets the suggested delay before the menu's `PopupMenu` is popped up or down
`void setMenuLocation (int x, int y)`	It sets the location of the pop-up component
`void setModel(ButtonModelnewModel)`	It sets the data model for the menu button
`void setPopupMenuVisible (boolean b)`	It sets the visibility of the menu's pop-up portion
`void setSelected(boolean b)`	It sets the selection status of the menu
`void updateUI()`	It is called by `UIFactory` when the look and feel has changed

When you have created menus by using the `JMenu` class, you populate those menus with menu items (see the next solution for the details).

Creating a Menu Item

"All right," says the NP, "I have created some menus. How do I add menu items to those menus—with the `JMenuItem` class?""You have got it," you say.

The actual items in Swing menus are supported by the `JMenuItem` class. Here's the inheritance diagram for this class:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.AbstractButton
                          |____javax.swing.JMenuItem
```

You'll find the constructors for the `JMenuItem` class in Table 16.6 and its methods in Table 16.7:

Table 16.6: Constructors of the JMenuItem class

Constructor	Does this
`JMenuItem()`	It constructs a `menu item`
`JMenuItem(Action a)`	It constructs a `menu item` whose properties are taken from the Action supplied
`JMenuItem(Icon icon)`	It constructs a `menu item` with an icon
`JMenuItem(String text)`	It constructs a `menu item` with text
`JMenuItem(String text, Icon icon)`	It constructs a `menu item` with the supplied text and icon
`JMenuItem(String text, int mnemonic)`	It constructs a `menu item` with the indicated text and keyboard mnemonic

Table 16.7: Methods of the JMenuItem class

Method	Does this
`protected void actionPropertyChanged(Action action, String propertyName)`	It updates the button's state in response to property changed in the associated action.
`void addMenuDragMouseListener (MenuDragMouseListener l)`	It adds a `MenuDragMouseListener` object
`void addMenuKeyListener(MenuKeyListener l)`	It adds a `MenuKeyListener` to the menu item
`protected void configurePropertiesFromAction(Action a)`	It sets the properties on this button to match those in the specified Action
`protected void fireMenuDragMouseDragged (MenuDragMouseEvent event)`	It fires the "mouse dragged" event
`protected void fireMenuDragMouseEntered (MenuDragMouseEvent event)`	It fires the "mouse entered" event
`protected void fireMenuDragMouseExited (MenuDragMouseEvent event)`	It fires the "mouse exited" event
`protected void fireMenuDragMouseReleased (MenuDragMouseEvent event)`	It fires the "mouse released" event
`protected void fireMenuKeyPressed (MenuKeyEvent event)`	It fires the "key pressed" event
`protected void fireMenuKeyReleased (MenuKeyEvent event)`	It fires the "key released" event

Table 16.7: Methods of the JMenuItem class

Method	Does this
`protected void fireMenuKeyTyped(MenuKeyEvent event)`	It fires the "key typed" event
`KeyStrokegetAccelerator()`	It gets the `KeyStroke` that serves as an accelerator for the menu item
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` object associated with this `JMenuItem`
`Component getComponent()`	It returns the `java.awt.Component` used to paint this object
`MenuDragMouseListener[] getMenuDragMouseListeners()`	It returns an array of all the `MenuDragMouseListeners` added to this `JMenuItem` with `addMenuDragMouseListener()`
`MenuKeyListener[] getMenuKeyListeners()`	It returns an array of all the `MenuKeyListeners` added to this `JMenuItem` with `addMenuKeyListener()`
`MenuElement[] getSubElements()`	It returns an array containing the submenu components for this menu component
`String getUIClassID()`	It gets the name of the `look and feel` class that renders this component
`protected void init(String text, Icon icon)`	It initializes the menu item with the indicated text and icon
`boolean isArmed()`	It determines whether the menu item is armed
`void menuSelectionChanged (booleanisIncluded)`	It is called by the `MenuSelectionManager` when the `MenuElement` is selected or unselected
`protected String paramString()`	It gets a string representation of this `JMenuItem`
`void processKeyEvent(KeyEvent e, MenuElement[] path, MenuSelectionManager manager)`	It processes a key event forwarded from the `MenuSelectionManager`
`void processMenuDragMouseEvent(MenuDragMouseEvent e)`	It handles a mouse drag in a menu
`void processMenuKeyEvent(MenuKeyEvent e)`	It handles a keystroke in a menu
`void processMouseEvent(MouseEvent e, MenuElement[] path, MenuSelectionManager manager)`	It processes a mouse event forwarded from the `MenuSelectionManager`
`void removeMenuDragMouseListener(MenuDragMouseListener l)`	It removes a `MenuDragMouseListener`
`void removeMenuKeyListener (MenuKeyListener l)`	It removes a `MenuKeyListener` from the menu item
`void setAccelerator(KeyStrokekeyStroke)`	It sets the key combination that invokes the menu item's action listeners without navigating the menu hierarchy
`void setArmed(boolean b)`	It identifies the menu item as armed
`void setEnabled(boolean b)`	It enables or disables the menu item
`void setModel(ButtonModelnewModel)`	It sets the model that this button represents
`void setUI(MenuItemUIui)`	It sets the look and feel object that renders this component
`void updateUI()`	It is called by `UIFactory` when the look and feel has changed

As you can see in Tables 16.6 and 16.7, the `JMenuItem` class has a lot to offer—for example, you can determine whether a menu item is "armed" (that is, it will be selected if the user releases the mouse) with the `isArmed()` and `setArmed()` methods.

This one and the previous two solutions have introduced menu bars, menus, and menu items in Swing; we'll put these elements together in the next solution to create a basic menu system.

Creating a Basic Menu System

"I have created a menu bar object, menu objects, and menu item objects," the NP says, "but how do I put them all together?" "Sit down and we'll go through it," you say, "it's not that hard."

The previous three solutions have introduced the classes needed to create menus in Swing programming: `JMenuBar`, `JMenu`, and `JMenuItem`. We'll put those classes to work in an example that displays two menus—File and Edit—in a menu bar. When the user clicks a menu item, the code will display which item was clicked in the applet's status bar.

We start by creating a menu bar with the `JMenuBar` class. Then we create the File menu with the `JMenu` class and three menu items for that menu—New, Open, and Exit—using the `JMenuItem` class:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
  CODE = Menu.class
  WIDTH = 350
  HEIGHT = 280 >
</APPLET>*/
public class Menu extends JApplet implements ActionListener
{
  public void init()
  {
  JMenuBar jmenubar = new JMenuBar();
  JMenu jmenu1 = new JMenu("File");
  JMenuItem jmenuitem1 = new JMenuItem("New..."),
  jmenuitem2 = new JMenuItem("Open..."),
  jmenuitem3 = new JMenuItem("Exit");
  ...
```

Now we can add the three menu items to the File menu by using the `JMenu()` class's `add()` method. Action listeners are used with menu items, so we'll give each menu item an action command and add an action listener to each item (note that menu separators can be added with the `addSeparator()` method):

```
public void init(){
JMenuBar jmenubar = new JMenuBar();
JMenu jmenu1 = new JMenu("File");
JMenuItem jmenuitem1 = new JMenuItem("New..."),
jmenuitem2 = new JMenuItem("Open..."),
jmenuitem3 = new JMenuItem("Exit");
jmenu1.add(jmenuitem1);
jmenu1.add(jmenuitem2);
jmenu1.addSeparator();
jmenu1.add(jmenuitem3);
jmenuitem1.setActionCommand("You selected New");
jmenuitem2.setActionCommand("You selected Open");
jmenuitem3.setActionCommand("You selected Exit");
jmenuitem1.addActionListener(this);
jmenuitem2.addActionListener(this);
jmenuitem3.addActionListener(this);
...
```

We can create the Edit menu in the same way, giving it three items—`Cut`, `Copy`, and `Paste`:

```
public void init(){
JMenuBar jmenubar = new JMenuBar();
JMenu jmenu1 = new JMenu("File");
JMenuItem jmenuitem1 = new JMenuItem("New..."),
jmenuitem2 = new JMenuItem("Open..."),
jmenuitem3 = new JMenuItem("Exit");
jmenu1.add(jmenuitem1);
jmenu1.add(jmenuitem2);
jmenu1.addSeparator();
jmenu1.add(jmenuitem3);
jmenuitem1.setActionCommand("You selected New");
```

```
    jmenuitem2.setActionCommand("You selected Open");
    jmenuitem3.setActionCommand("You selected Exit");
    jmenuitem1.addActionListener(this);
    jmenuitem2.addActionListener(this);
    jmenuitem3.addActionListener(this);
    JMenu jmenu2 = new JMenu("Edit");
    JMenuItem jmenuitem4 = new JMenuItem("Cut"),
    jmenuitem5 = new JMenuItem("Copy"),
    jmenuitem6 = new JMenuItem("Paste");
    jmenu2.add(jmenuitem4);
    jmenu2.add(jmenuitem5);
    jmenu2.add(jmenuitem6);
    jmenuitem4.setActionCommand("You selected Cut");
    jmenuitem5.setActionCommand("You selected Copy");
    jmenuitem6.setActionCommand("You selected Paste");
    jmenuitem4.addActionListener(this);
    jmenuitem5.addActionListener(this);
    jmenuitem6.addActionListener(this);
    ...
    }
```

All that's left is to use the `JMenuBar` class's `add()` method to add the menus to the menu bar and to add the menu bar itself to the applet with the `setJMenuBar()` method:

```
    public void init(){
    JMenuBar jmenubar = new JMenuBar();
    JMenu jmenu1 = new JMenu("File");
    JMenuItem jmenuitem1 = new JMenuItem("New..."),
    jmenuitem2 = new JMenuItem("Open..."),
    jmenuitem3 = new JMenuItem("Exit");
    jmenu1.add(jmenuitem1);
    jmenu1.add(jmenuitem2);
    jmenu1.addSeparator();
    jmenu1.add(jmenuitem3);
    jmenuitem1.setActionCommand("You selected New");
    jmenuitem2.setActionCommand("You selected Open");
    jmenuitem3.setActionCommand("You selected Exit");
    jmenuitem1.addActionListener(this);
    jmenuitem2.addActionListener(this);
    jmenuitem3.addActionListener(this);
    JMenu jmenu2 = new JMenu("Edit");
    JMenuItem jmenuitem4 = new JMenuItem("Cut"),
    jmenuitem5 = new JMenuItem("Copy"),
    jmenuitem6 = new JMenuItem("Paste");
    jmenu2.add(jmenuitem4);
    jmenu2.add(jmenuitem5);
    jmenu2.add(jmenuitem6);
    jmenuitem4.setActionCommand("You selected Cut");
    jmenuitem5.setActionCommand("You selected Copy");
    jmenuitem6.setActionCommand("You selected Paste");
    jmenuitem4.addActionListener(this);
    jmenuitem5.addActionListener(this);
    jmenuitem6.addActionListener(this);
    jmenubar.add(jmenu1);
    jmenubar.add(jmenu2);
    setJMenuBar(jmenubar);
    }
```

The menu system is now displayed to the user. When the user clicks a menu item, we'll get the item's action command and display it in the applet's status bar, like this, by using the `actionPerformed()` method. Here's the code:

```
    public void actionPerformed(ActionEvent e){
    JMenuItem jmenuitem = (JMenuItem)e.getSource();
    showStatus(jmenuitem.getActionCommand());
    }
    }
```

That's all to create a basic menu system. Working with menus in Swing is much like working with them in AWT, with the obvious difference that applet windows in AWT can't display menus. The result of this code appears as shown in Figure 16.1, where you can see the menu system at work.

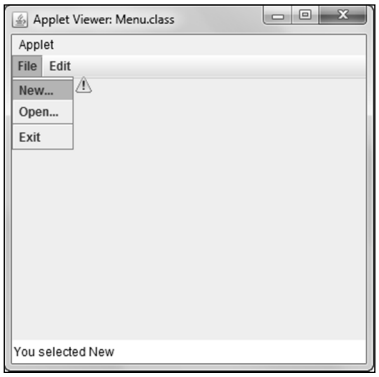


Figure 16.1: A basic menu system

Adding Images to Menu Items

The Aesthetic Design Coordinator says, "Your menu items are so bland—can't you jazz things up a bit?" "Well," you say, "I can add images to them." "Great!" says the ADC, "I'll send you the images I want to use." "Um," you say.

It's simple to add images to menu items in Swing programming; you just add an icon to the appropriate constructor call. Here's how we add an image to each menu item in the example from the previous solution:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
  CODE = MenuImages.class
  WIDTH = 350
  HEIGHT = 280 >
</APPLET>*/
public class MenuImages extends JApplet implements ActionListener {
ImageIcon icon = new ImageIcon("blug2.jpg");
ImageIcon icon1 = new ImageIcon("cut.jpg");
ImageIcon icon2 = new ImageIcon("copy.jpg");
ImageIcon icon3 = new ImageIcon("paste.jpg");
public void init() {
JMenuBar jmenubar = new JMenuBar();
JMenu jmenu1 = new JMenu("File");
JMenuItem jmenuitem1 = new JMenuItem("New...", icon),
jmenuitem2 = new JMenuItem("Open...", icon),
jmenuitem3 = new JMenuItem("Exit", icon);
jmenu1.add(jmenuitem1);
jmenu1.add(jmenuitem2);
jmenu1.addSeparator();
jmenu1.add(jmenuitem3);
jmenuitem1.setActionCommand("You selected New");
jmenuitem2.setActionCommand("You selected Open");
jmenuitem3.setActionCommand("You selected Exit");
jmenuitem1.addActionListener(this);
jmenuitem2.addActionListener(this);
jmenuitem3.addActionListener(this);
JMenu jmenu2 = new JMenu("Edit");
JMenuItem jmenuitem4 = new JMenuItem("Cut", icon1),
jmenuitem5 = new JMenuItem("Copy", icon2),
jmenuitem6 = new JMenuItem("Paste", icon3);
```

```
    jmenu2.add(jmenuitem4);
    jmenu2.add(jmenuitem5);
    jmenu2.add(jmenuitem6);
    jmenuitem4.setActionCommand("You selected Cut");
    jmenuitem5.setActionCommand("You selected Copy");
    jmenuitem6.setActionCommand("You selected Paste");
    jmenuitem4.addActionListener(this);
    jmenuitem5.addActionListener(this);
    jmenuitem6.addActionListener(this);
    jmenubar.add(jmenu1);
    jmenubar.add(jmenu2);
    setJMenuBar(jmenubar);
    setJMenuBar(jmenubar);
     }
      public void actionPerformed(ActionEvent e) {
      JMenuItem jmenuitem = (JMenuItem)e.getSource();
      showStatus(jmenuitem.getActionCommand()); }
    }
```

The result appears as shown in Figure 16.2:

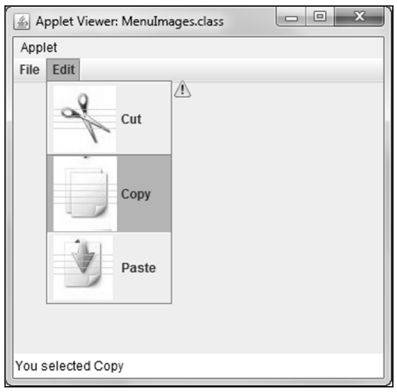


Figure 16.2: Adding images to menu items

As you can see in the figure, each menu item now displays an image.

Creating Checkbox Menu Items

The Product Support Specialist appears and says, "Users are unhappy with your new program, especially the Make All Text Invisible menu item, because they never know whether that option is enabled until it's too late." "OK," you say. "I'll add a checkmark in front of the item to show when it's active."

Swing menus support checkbox menu items just as AWT menus do. You use the `JcheckBoxMenuItem` class to create checkbox menu items in Swing programming. Here's the inheritance diagram for this class:

```
java.lang.Object
   |____java.awt.Component
         |____java.awt.Container
                |____javax.swing.JComponent
                       |____javax.swing.AbstractButton
                              |____javax.swing.JMenuItem
                                     |____javax.swing.JCheckBoxMenuItem
```

You'll find the constructors for the `JCheckBoxMenuItem` class in Table 16.8 and its methods in Table 16.9:

Table 16.8: Constructors of the JCheckBoxMenuItem class

Constructor	Does this
`JcheckBoxMenuItem()`	It constructs an unselected checkbox menu item
`JCheckBoxMenuItem(Action a)`	It creates a menu item whose properties are taken from the action supplied

Table 16.8: Constructors of the JCheckBoxMenuItem class	
Constructor	**Does this**
`JcheckBoxMenuItem(Icon icon)`	It constructs an initially unselected check box menu item with an icon
`JcheckBoxMenuItem(String text)`	It constructs an initially unselected check box menu item with text
`JCheckBoxMenuItem(String text, boolean b)`	It constructs a check box menu item with the indicated text and the selection state
`JCheckBoxMenuItem(String text, Icon icon)`	It constructs an initially unselected check box menu item with the indicated text and icon
`JCheckBoxMenuItem(String text, Icon icon, boolean b)`	It constructs a check box menu item with the indicated text, icon, and selection state

Table 16.9: Methods of the JCheckBoxMenuItem class	
Method	**Does this**
`AccessibleContextgetAccessibleContext()`	It gets the `AccessibleContext` object
`Object[] getSelectedObjects()`	It gets an array (length 1) containing the selected checkbox menu item's label
`booleangetState()`	It gets the selected state of the item
`String getUIClassID()`	It gets the name of the `look and feel` class that renders this component
`protected String paramString()`	It gets a string representation of this `JCheckBoxMenuItem` object
`void setState(boolean b)`	It sets the selected state of the item

Let's look at an example. In this case, we add four checkbox items to a menu by using the `JCheckBoxMenuItem` class. Here's how we create and add those items to a menu (note that action listeners can be used with checkbox menu items):

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
  CODE = MenuCheckBox.class
  WIDTH = 350
  HEIGHT = 280 >
</APPLET>*/
public class MenuCheckBox extends JApplet implements ActionListener {
JCheckBoxMenuItem
jcheckboxmenuitem1 = new JCheckBoxMenuItem("MBA"),
jcheckboxmenuitem2 = new JCheckBoxMenuItem("MCA"),
jcheckboxmenuitem3 = new JCheckBoxMenuItem("BTECH"),
jcheckboxmenuitem4 = new JCheckBoxMenuItem("MTECH");
public void init() {
Container contentPane = getContentPane();
JMenuBar jmbar = new JMenuBar();
JMenu jmenu = new JMenu("COURSES");
jcheckboxmenuitem1.addActionListener(this);
jcheckboxmenuitem2.addActionListener(this);
jcheckboxmenuitem3.addActionListener(this);
jcheckboxmenuitem4.addActionListener(this);
jmenu.add(jcheckboxmenuitem1);
jmenu.add(jcheckboxmenuitem2);
jmenu.add(jcheckboxmenuitem3);
jmenu.add(jcheckboxmenuitem4);
jmbar.add(jmenu);
```

```
    setJMenuBar(jmbar);
    }

       ...
    }
```

When the user clicks a checkbox item, we list the state of all four items as follows:

```
    public void actionPerformed(ActionEvent e) {
    showStatus("MBA: " + jcheckboxmenuitem1.getState() +
    " MCA: " + jcheckboxmenuitem2.getState() +
    " BTECH: " + jcheckboxmenuitem3.getState() +
    " MTECH: " + jcheckboxmenuitem4.getState());
    }
```

That's all it takes. The result of this code appears as shown in Figure 16.3:

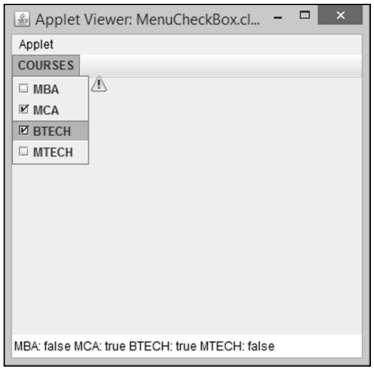


Figure 16.3: Creating checkbox menu items

As you can see in the figure, the checkboxes appear as they should be in the menu.

Creating Radio Button Menu Items

"OK," says the NP, "I know you create checkbox menu items with the `JCheckBoxMenuItem` class. But what about radio buttons? Don't you have to add checkbox items to a group?" "Almost right," you say. "In fact, you add the `JradioButtonMenuItem` objects to a group."

You use the `JradioButtonMenuItem` class to create radio button menu items in Swing. Here's the inheritance diagram for this class:

```
    java.lang.Object
      |____java.awt.Component
            |____java.awt.Container
                  |____javax.swing.JComponent
                        |____javax.swing.AbstractButton
                              |____javax.swing.JMenuItem
                                    |____javax.swing.JRadioButtonMenuItem
```

You'll find the constructors for the `JradioButtonMenuItem` class in Table 16.10 and its methods in Table 16.11:

Table 16.10: Constructors of the JRadioButtonMenuItem class

Constructor	Does this
`JRadioButtonMenuItem()`	It constructs a `JRadioButtonMenuItem`
`JRadioButtonMenuItem(Action a)`	It creates a radio button menu item whose properties are taken from the action supplied
`JRadioButtonMenuItem(Icon icon)`	It constructs a `JRadioButtonMenuItem` with an icon
`JRadioButtonMenuItem(Icon icon, boolean selected)`	It constructs a `JRadioButtonMenuItem` with the indicated image and selection state, but no text

Table 16.10: Constructors of the JRadioButtonMenuItem class	
Constructor	**Does this**
`JRadioButtonMenuItem(String text)`	It constructs a `JRadioButtonMenuItem` with text
`JRadioButtonMenuItem(String text, booleanselected)`	It constructs a radio button menu item with the indicated text and the selection state
`JRadioButtonMenuItem(String text, Icon icon)`	It constructs a `JRadioButtonMenuItem` with the indicated text and icon
`JRadioButtonMenuItem(String text, Icon icon, boolean selected)`	It constructs a radio button menu item that has the indicated text, image, and selection state

Table 16.11: Methods of the JRadioButtonMenuItem class	
Method	**Does this**
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` object
`String getUIClassID()`	It gets the name of the look and feel class that renders this component
`protected String paramString()`	It gets a string representation of this `JRadioButtonMenuItem`

Let's look at an example. In this case, we add four radio buttons to a menu and add an item listener to indicate when each radio button is selected or deselected. Here's what the code looks like:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
  CODE = MenuRadioButton.class
  WIDTH = 350
  HEIGHT = 280 >
</APPLET>*/
public class MenuRadioButton extends JApplet implements ItemListener {
  JRadioButtonMenuItem
  jradiobuttonmenuitem1 = new JRadioButtonMenuItem("Uttar Pradesh"),
  jradiobuttonmenuitem2 = new JRadioButtonMenuItem("Delhi"),
  jradiobuttonmenuitem3 = new JRadioButtonMenuItem("Assam"),
  jradiobuttonmenuitem4 = new JRadioButtonMenuItem("Chennai");
  public void init() {
        Container contentPane = getContentPane();
        JMenuBar jmenubar = new JMenuBar();
        JMenu jmenu = new JMenu("Radio Button Menu Items");
        jmenu.add(jradiobuttonmenuitem1);
        jmenu.add(jradiobuttonmenuitem2);
        jmenu.add(jradiobuttonmenuitem3);
        jmenu.add(jradiobuttonmenuitem4);
        ButtonGroup group = new ButtonGroup();
        group.add(jradiobuttonmenuitem1);
        group.add(jradiobuttonmenuitem2);
        group.add(jradiobuttonmenuitem3);
        group.add(jradiobuttonmenuitem4);
        jradiobuttonmenuitem1.addItemListener(this);
        jradiobuttonmenuitem2.addItemListener(this);
        jradiobuttonmenuitem3.addItemListener(this);
        jradiobuttonmenuitem4.addItemListener(this);
        jmenubar.add(jmenu);
        setJMenuBar(jmenubar); }
  public void itemStateChanged(ItemEvent e) {
        JMenuItem jmenuitem = (JMenuItem) e.getSource();
        String itemtext = jmenuitem.getText();
        if(e.getStateChange() == ItemEvent.SELECTED)
        itemtext += " was selected";
  else
        itemtext += " was deselected";
```

```
      showStatus(itemtext); }
  }
```

The result of this code appears as shown in Figure 16.4, where you can see the radio buttons in the menu at work:

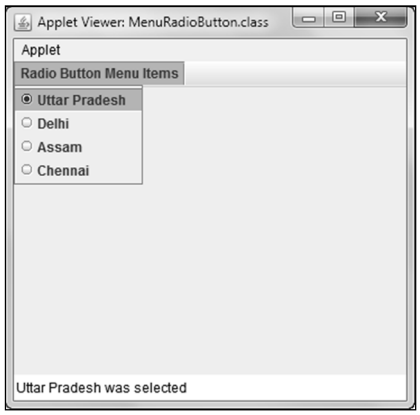


Figure 16.4: Creating radio button menu items

When the user clicks a radio button, this applet indicates which one was selected in the status bar.

Creating Submenus

"My menus are getting too big," the NP says. "I want to let the user select the drawing color in my program by selecting menu items, but all the colors are crowding out the other items in the menu." "You should try putting all the colors into a submenu," you say. "How many colors do you want to display?" "About 3,000," says the NP. You say, "Uh-oh."

Submenus, also called *cascading menus*, are menus you open from other menus. Swing indicates that a menu item is actually a submenu by adding a clickable arrow to the right of the menu item; when the user clicks the arrow, the submenu opens.

You can create submenus easily in Swing. All you do to create a new menu is add items to the menu and then add the menu itself as an item in another menu. Here's an example to make this clearer. In this case, we add four menu items and three submenu items. Here's what this looks like in code:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
  CODE = SubMenus.class
  WIDTH = 350
  HEIGHT = 280 >
</APPLET>*/
public class SubMenus extends JApplet
{
  public void init()
  {
        JMenuBar jmenubar = new JMenuBar();
        JMenu jm = new JMenu("Sub Menus", true);
        JMenu jsubmenu = new JMenu("Cascading Menu", true);
        jm.add("Item 1");
        jm.add("Item 2");
        jm.add("Item 3");
        jsubmenu.add("Sub Item a");
        jsubmenu.add("Sub Item b");
        jsubmenu.add("Sub Item c");
```

```
            jm.add(jsubmenu);
            jmenubar.add(jm);
            setJMenuBar(jmenubar);
        }
    }
```

The result of this code appears, as shown in Figure 16.5:

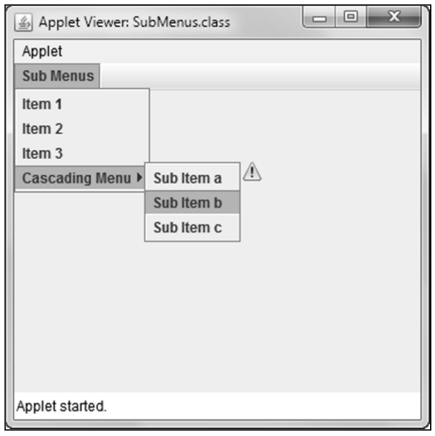


Figure 16.5: Creating submenus

As you can see, the submenu opens when the user selects the corresponding menu item.

Creating Menu Accelerators

"Say," says the Programming Correctness Czar, "what about those menu items I sometimes see that have keyboard shortcuts? But you can't do that in Java." "Sure can," you say, "with menu accelerators."

Menu *accelerators* provide keyboard access to menu items.

In accelerators, you specify the actual keystrokes involved to trigger a menu item, such as F1, Ctrl+X, and so on (the meta key isn't used in menu accelerators unless you specify that it should be). You can add an accelerator to a menu item (not menus) with the `setAccelerator()` method, which takes an object of the `KeyStroke` class that defines the keystroke if you want to use as the accelerator. Here's the inheritance diagram for the `KeyStroke` class:

```
java.lang.Object
  |____java.awt.AWTKeyStroke
        |________javax.swing.KeyStroke
```

You can find the methods of the `KeyStroke` class in Table 16.12:

Table 16.12: Methods of the KeyStroke class

Method	Does this
`static KeyStroke getKeyStroke(char keyChar)`	It returns a shared instance of a keystroke that's activated when the key is pressed.
`static KeyStroke getKeyStroke(char keyChar, booleanonKeyRelease)`	These are deprecated and they use `getKeyStroke(char)`.
`static KeyStroke getKeyStroke(intkeyCode, int modifiers)`	It returns a shared instance of a keystroke given a char code and a set of modifiers. The key is activated when it's pressed.
`static KeyStroke getKeyStroke(intkeyCode, int modifiers, booleanonKeyRelease)`	It gets a shared instance of a keystroke given a numeric `keycode` and a set of modifiers.
`static KeyStroke getKeyStroke(String s)`	It parses a string and returns a `KeyStroke`.
`static KeyStroke getKeyStrokeForEvent(KeyEventanEvent)`	It returns a keystroke from an event.

You can use the virtual key constants defined in the `KeyEvent`class—`KeyEvent.VK_A`,`KeyEvent.VK_ENTER`, and `KeyEvent.VK_TAB`—to specify the keycode when creating a `KeyStroke` object. The modifiers you can specify can be any combination of the following:

- InputEvent.SHIFT_DOWN_MASK
- InputEvent.CTRL_DOWN_MASK
- InputEvent.META_DOWN_MASK
- InputEvent.ALT_DOWN_MASK
- InputEvent.ALT_GRAPH_DOWN_MASK

We'll put this together in an example. In this case, we add a mnemonic, the letter `C` to the Copyitem in the Edit menu and give it the accelerator Ctrl+C. Here's how that looks in code:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
   CODE = MenuAccelerator.class
   WIDTH = 350
   HEIGHT = 280 >
</APPLET>*/
public class MenuAccelerator extends JApplet implements ActionListener {
public void init() {
Container contentPane = getContentPane();
JMenuBar jmenubar = new JMenuBar();
JMenu jmenu = new JMenu("Edit");
JMenuItem jmenuitem = new JMenuItem("Copy...");
jmenu.add(jmenuitem);
jmenu.add("Open ...");
jmenuitem.setMnemonic(KeyEvent.VK_C);
KeyStroke keystroke = KeyStroke.getKeyStroke(KeyEvent.VK_C,
Event.CTRL_MASK);
jmenuitem.setAccelerator(keystroke);
jmenuitem.addActionListener(this);
jmenubar.add(jmenu);
setJMenuBar(jmenubar); }
public void actionPerformed(ActionEvent e) {
   showStatus("You selected the Copy item."); }
}
```

The result is shown in Figure 16.6, where you see the accelerator (Ctrl+C) in the Copy menu item:

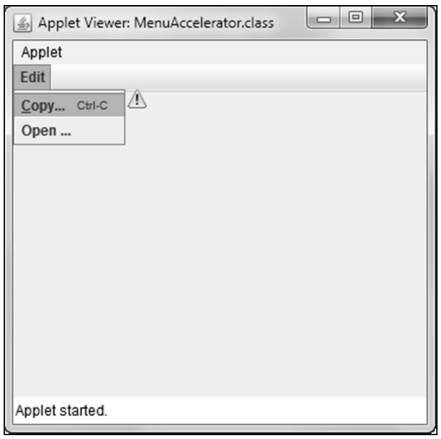


Figure 16.6: Menu mnemonics

Providing accelerators like this can speed up things for users who don't want to switch from the keyboard to the mouse and back again.

Enabling/Disabling and Changing Menu Items at Runtime

"That Johnson," the NP says, "clicked the wrong option in my program at the wrong time, which made my code try to check email when it wasn't even connected to the Internet!" "Silly Johnson," you say. "How about disabling menu items when they are not appropriate?" "Perfect!" says the NP.

You can enable and disable menu items with the `setEnabled()` method; when a menu item is disabled, it appears dimmed and cannot be clicked. Here's a quick example: In this case, we let the user disable a menu item in an applet's Edit menu. To do that, we add two items to the Edit menu—Disable bottom item and Enable bottom item. When the user clicks Disable bottom item, we disable a third menu item in the Edit menu and switch its caption to "Disabled item" with the `setText()` method; when the user clicks Enable bottom item, we enable the third item and switch its caption to "Enabled item". Here's what this looks like in code:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
  CODE = MenuDisable.class
  WIDTH = 350
  HEIGHT = 280 >
</APPLET>*/
public class MenuDisable extends JApplet implements ActionListener {
JMenuBar jmenubar = new JMenuBar();
JMenu menu1 = new JMenu("File");
JMenu menu2 = new JMenu("Edit");
JMenuItem jmenuitem1 = new JMenuItem("Save..."),
jmenuitem2 = new JMenuItem("Save As..."),
jmenuitem3 = new JMenuItem("Exit"),
jmenuitem4 = new JMenuItem("Disable bottom item"),
jmenuitem5 = new JMenuItem("Enable bottom item"),
jmenuitem6 = new JMenuItem("Enabled item");

public void init() {
menu1.add(jmenuitem1);
menu1.add(jmenuitem2);
menu1.addSeparator();
menu1.add(jmenuitem3);
jmenuitem1.addActionListener(this);
jmenuitem2.addActionListener(this);
jmenuitem3.addActionListener(this);

menu2.add(jmenuitem4);
menu2.add(jmenuitem5);
menu2.add(jmenuitem6);
jmenuitem4.addActionListener(this);
jmenuitem5.addActionListener(this);
jmenuitem6.addActionListener(this);

jmenubar.add(menu1);
jmenubar.add(menu2);
setJMenuBar(jmenubar); }

  public void actionPerformed(ActionEvent e) {
        JMenuItem jmenuitem = (JMenuItem)e.getSource();
        if(jmenuitem == jmenuitem4) {
        jmenuitem6.setText("Disabled item");
        jmenuitem6.setEnabled(false); }
           if(jmenuitem == jmenuitem5) {
        jmenuitem6.setText("Enabled item");
        jmenuitem6.setEnabled(true); }
        }
}
```

You can see the result of this code in Figure 16.7:

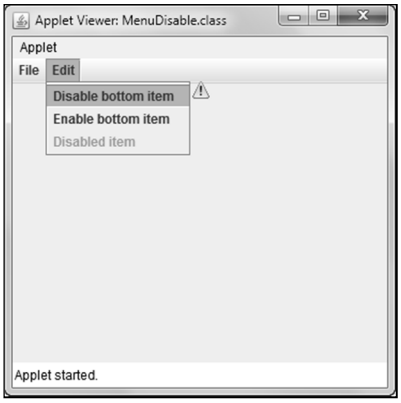


Figure 16.7: Disabling menu items

When the user selects an item in the Edit menu, the bottom item in that menu is enabled or disabled accordingly, and its caption is changed to match. That's all there is to it.

TIP

You shouldn't disable too many menu items at once—doing so gives your program an inaccessible feel.

Adding and Removing Menu Items at Runtime

The Big Boss says, "We are providing the options available in our new program on the user's ability to pay." You say, "What?" "After we have determined how much money the user has, we'll enable the appropriate number of menu items." You say, "What?"

You can add menu items to menus at runtime with the `add()` method and remove them with the `remove()` method. Here's an example in which we add a new menu item to a menu when the user clicks "Add item," and we remove it when the user clicks "Remove item." Here's what the code looks like:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
  CODE = MenuUpdate.class
  WIDTH = 350
  HEIGHT = 280 >
</APPLET>*/
public class MenuUpdate extends JApplet implements ActionListener {
  JMenuBar jmenubar = new JMenuBar();
  JMenu menu1 = new JMenu("File");
  JMenu menu2 = new JMenu("Format");
  JMenuItem jmenuitem1 = new JMenuItem("New..."),
  jmenuitem2 = new JMenuItem("Open..."),
  jmenuitem3 = new JMenuItem("Exit"),
  jmenuitem4 = new JMenuItem("Add item"),
  jmenuitem5 = new JMenuItem("Remove item"),
  jmenuitem6 = new JMenuItem("New item");
  public void init() {
        menu1.add(jmenuitem1);
        menu1.add(jmenuitem2);
        menu1.addSeparator();
        menu1.add(jmenuitem3);
        jmenuitem1.addActionListener(this);
        jmenuitem2.addActionListener(this);
        menu2.add(jmenuitem4);
```

```
            menu2.add(jmenuitem5);
            jmenuitem4.setActionCommand("You selected Font");
            jmenuitem4.addActionListener(this);
            jmenuitem5.setActionCommand("You selected Font");
            jmenuitem5.addActionListener(this);
            jmenubar.add(menu1);
            jmenubar.add(menu2);
            setJMenuBar(jmenubar); }
            public void actionPerformed(ActionEvent e) {
                    JMenuItem jmenuitem = (JMenuItem)e.getSource();
                    if(jmenuitem == jmenuitem4) { menu2.add(jmenuitem6); }
                    if(jmenuitem == jmenuitem5) { menu2.remove(jmenuitem6); }
            }
    }
```

The result of this code appears as shown in Figure 16.8, where you can see the new item added when the user clicks Add item:

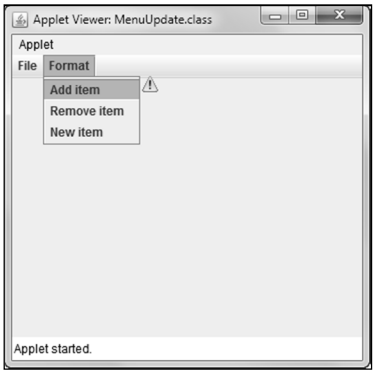


Figure 16.8: Adding and removing menu items at runtime

The user can remove the new item by clicking the Remove item. Note that you can create new menu items on the fly with the new operator, and you can add as many menu items to your menus as you like at runtime.

Adding Buttons and Other Controls to Menus

You can add controls, such as buttons, to Swing menus. In fact, doing so is easy; you just use the `JMenu` class's `add()` method. For example, here's how we add a `Jbutton` object to a menu, making the code display a message "Hello User!" when the user clicks the button:

```
    import java.awt.*;
    import javax.swing.*;
    import java.awt.event.*;
    /* <APPLET
      CODE = MenuControls.class
      WIDTH = 350
      HEIGHT = 280 >
    </APPLET>*/
    public class MenuControls extends JApplet implements ActionListener {
    public void init() {
    JMenuBar jmenubar = new JMenuBar();
    JMenu menu1 = new JMenu("File");
    JMenuItem jmenuitem1 = new JMenuItem("New..."),
    jmenuitem2 = new JMenuItem("Save..."),
    jmenuitem3 = new JMenuItem("Save As…");
    JButton jbutton = new JButton("Click Here!");
    jbutton.setActionCommand("Hello User! ");
    jbutton.addActionListener(this);
    menu1.add(jmenuitem1);
    menu1.add(jmenuitem2);
```

```
    menu1.addSeparator();
    menu1.add(jbutton);
    menu1.addSeparator();
    menu1.add(jmenuitem3);
    JMenu jmenu2 = new JMenu("Edit");
    JMenuItem jmenuitem4 = new JMenuItem("Cut"),
    jmenuitem5 = new JMenuItem("Copy"),
    jmenuitem6 = new JMenuItem("Paste");
    jmenu2.add(jmenuitem4);
    jmenu2.add(jmenuitem5);
    jmenu2.add(jmenuitem6);
    jmenubar.add(menu1);
    jmenubar.add(jmenu2);
    setJMenuBar(jmenubar); }
    public void actionPerformed(ActionEvent e) {
    JButton jbutton = (JButton)e.getSource();
    showStatus(jbutton.getActionCommand()); }
    }
```

The result of this code appears in Figure 16.9:

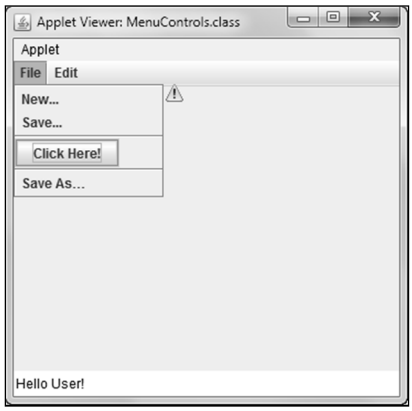


Figure 16.9: Adding a button to a menu

As you can see in the figure, we have added a completely functional button to the File menu of the applet.

Creating Pop-Up Menus

The Big Boss is annoyed and says, "All these new features to keep up with, all these programmers to hire! The expense! Now it's pop-up menus, no less!" "That's OK," you say. "I can add those to our programs if I just put in a little overtime." The BB yells, "The expense! The expense!"

Pop-up menus are the menus that the user can display by clicking the mouse (usually the right mouse button) when it's positioned over a component. In Swing, pop-up menus are supported by the (you guessed it) `JPopupMenu` class. Here's the inheritance diagram for that class:

```
    java.lang.Object
      |____java.awt.Component
            |____java.awt.Container
                    |____javax.swing.JComponent
                            |____javax.swing.JPopupMenu
```

You'll find the constructors for the `JPopupMenu` class in Table 16.13 and its methods in Table 16.14:

Table 16.13: Constructors of the JPopupMenu class

Constructor	Does this
`JPopupMenu()`	It constructs a `JPopupMenu`
`JPopupMenu(String label)`	It constructs a `JPopupMenu` with the indicated title

Table 16.14: Methods of the JPopupMenu class

Method	Does this
`JMenuItem add(Action a)`	It appends a new menu item to the end of the menu, which initiates the indicated action
`JMenuItem add(JMenuItem menuItem)`	It appends the indicated menu item
`JMenuItem add(String s)`	It constructs a new item with the indicated text and appends it to the end of the menu
`void addPopupMenuListener(PopupMenuListener l)`	It adds a `PopupMenu` listener
`void addSeparator()`	It appends a new separator at the end of the menu
`protected PropertyChangeListener createActionChangeListener(JMenuItem b)`	It creates an action changed listener
`protected void firePopupMenuCanceled()`	It informs listeners that this pop-up menu is cancelled
`protected void firePopupMenuWillBecomeInvisible()`	It informs `PopupMenuListeners` that this pop-up menu will become invisible
`protected void firePopupMenuWillBecomeVisible()`	It informs `PopupMenuListeners` that this pop-up menu will become visible
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` object
`Component getComponent()`	It gets the `JPopupMenu` used to paint the receiving element
`Component Container.getComponent(int i)`	It gets the component at the indicated index
`int getComponentIndex(Component c)`	It gets the index of the indicated component
`static boolean getDefaultLightWeightPopupEnabled()`	It returns the default value for the `defaultLightWeightPopupEnabled` property
`Component getInvoker()`	It gets the component that's the "invoker" of this pop-up menu (that is, the component in which the pop-up menu is to be displayed)
`String getLabel()`	It gets the pop-up menu's label
`Insets getMargin()`	It gets the margin between the pop-up menu's border and its contained components
`SingleSelectionModel getSelectionModel()`	It gets the model object that handles single selections
`MenuElement[] getSubElements()`	It should return an array containing the MenuElements
`PopupMenuUI getUI()`	It gets the `look and feel` object that renders this component
`String getUIClassID()`	It gets the name of the `look and feel` class that renders this component
`void insert(Action a, int index)`	It inserts a menu item for the indicated `Action` object at a given position
`void insert(Component component, int index)`	It inserts the indicated component into the menu at a given position
`boolean isBorderPainted()`	It checks whether the border should be painted
`boolean isLightWeightPopupEnabled()`	It returns True if lightweight pop-ups are in use and false if heavyweight pop-ups are in use
`boolean isVisible()`	It returns True if the pop-up menu is visible
`void menuSelectionChanged(booleanisIncluded)`	It is called when the menu selection is changed

Table 16.14: Methods of the JPopupMenu class	
Method	**Does this**
`void pack()`	It lays out the container so that it uses the minimum space to display its contents
`protected void paintBorder(Graphics g)`	It paints the pop-up menu's border
`protected String paramString()`	It gets a string representation of this
`void processKeyEvent(KeyEvent e, MenuElement[]path, MenuSelectionManager manager)`	It processes a key event
`void processMouseEvent(MouseEvent event, MenuElement[] path, MenuSelectionManager manager)`	It processes a mouse event
`void remove(int pos)`	It removes the component at the indicated index
`void removePopupMenuListener(PopupMenuListener l)`	It removes a `PopupMenu` listener
`void setBorderPainted(boolean b)`	It sets whether the border should be painted
`static void setDefaultLightWeightPopupEnabled(boolean aFlag)`	It sets the default value for the `lightWeightPopupEnabled` property
`void setInvoker(Component invoker)`	It sets the invoker of this pop-up menu
`void setLabel(String label)`	It sets the pop-up menu's label
`void setLightWeightPopupEnabled (boolean aFlag)`	It chooses whether to use a lightweight pop-up if it fits
`void setLocation(int x, int y)`	It sets the location of the upper-left corner of the pop-up menu
`void setPopupSize(Dimension d)`	It sets the size of the pop-up by using a `Dimension` object
`void setPopupSize(int width, int height)`	It sets the size of the pop-up
`void setSelected(Component sel)`	It sets the currently selected component
`void setSelectionModel(SingleSelectionModel model)`	It sets the model object to handle single selections
`void setUI(PopupMenuUI ui)`	It sets the `look and feel` object that draws this component
`void setVisible(boolean b)`	It sets the visibility of the menu
`void show(Component invoker, int x, int y)`	It displays the pop-up menu
`void updateUI()`	It is called by `UIFactory` when the look and feel has changed

You create pop-up menus, add new items to them with the `add()` method, and then display them with the `show()` method.

Let's look at an example. In this case, we create a pop-up menu with three items in it—`Find`, `Find Next`, and `Replace`–which will appear when the user right-clicks with the mouse. A pop-up menu needs to be the child of some other component, so we'll add a text field to an applet and cover the applet with the text field. Here's what creating the text field and the pop-up menu looks like in code:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
   CODE = PopUp.class
   WIDTH = 350
```

```
    HEIGHT = 280 >
</APPLET>*/
public class PopUp extends JApplet implements MouseListener
{
JTextField jtextfield = new JTextField("Right click me!", JTextField.CENTER);
JPopupMenu jpopupmenu = new JPopupMenu();
public void init()
{
Container contentPane = getContentPane();
jpopupmenu.add(new JMenuItem("Find", new ImageIcon("desert.jpg")));
jpopupmenu.add(new JMenuItem("Find Next", new ImageIcon("desert.jpg")));
jpopupmenu.add(new JMenuItem("Replace", new ImageIcon("desert.jpg")));
jtextfield.addMouseListener(this);
contentPane.add(jtextfield);
}

...

}
```

Now when the user clicks a mouse button, we can check whether the right mouse button was clicked by using the `MouseEvent`class's `getModifiers()` method and the mask for the right mouse button—`InputEvent.BUTTON3_MASK`. If the right mouse button was indeed clicked, we'll show the new pop-up menu at the mouse location, like this:

```
public void mousePressed (MouseEvent e)
{
        if((e.getModifiers() & InputEvent.BUTTON3_MASK) == InputEvent.BUTTON3_MASK)
        jpopupmenu.show(jtextfield, e.getX(), e.getY());
}
public void mouseClicked(MouseEvent e) {}
public void mouseReleased(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
```

The result appears as shown in Figure 16.10:

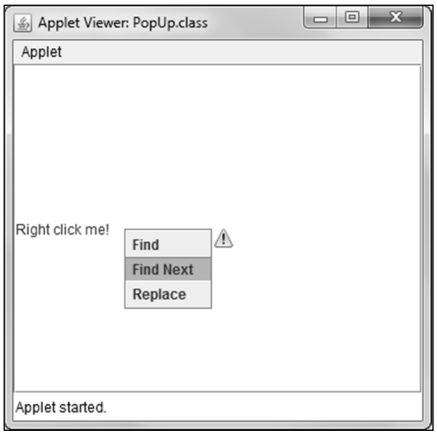


Figure 16.10: Displaying a pop-up menu

When the user right-clicks the text field in the applet, the pop-up menu appears at the mouse location (Figure 16.10). That's all there is to display pop-up menus.

Creating Toolbars

"Menus are fine," says the NP, "but sometimes I'm working too fast to use them, so I have got a lot of menu accelerators in my program. But now I keep forgetting them." You smile and say, "Have you tried adding a toolbar?"

Toolbars display buttons and other controls that represent common actions in your program, such as saving a file or pasting the contents of the clipboard. Buttons in a toolbar often represent frequently used menu items in your menu system. In Swing, you use the `JToolBar` class to create toolbars. Here's the inheritance diagram for the `JToolBar` class:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.JToolBar
```

You can find the constructors for the `JToolBar`class in Table 16.15 and its methods in Table 16.16:

Table 16.15: Constructors of the JToolBar class

Constructor	Does this
`JToolBar()`	It constructs a new toolbar
`JToolBar(int orientation)`	It constructs a new toolbar with the given orientation
`JToolBar(String name)`	It creates a new tool bar with the specified name
`JToolBar(String name, int orientation)`	It creates a new tool bar with a specified name and orientation

Table 16.16: Methods of the JToolBarclass

Method	Does this
`JButton add(Action a)`	It adds a new `JButton` that initiates the action
`protected void addImpl(Component comp, Object constraints, int index)`	It adds the indicated component to this container at the indicated index
`void addSeparator()`	It appends a toolbar separator
`void addSeparator(Dimension size)`	It appends a toolbar separator with the given dimensions
`protected PropertyChangeListener createActionChangeListener(JButton b)`	It creates an action changed listener
`protected JButton createActionComponent (Action a)`	It creates the `JButton` for actions added to the `JToolBar`
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext`
`Component getComponentAtIndex(int i)`	It gets the component at the indicated index
`int getComponentIndex(Component c)`	It gets the index of the indicated component
`Insets getMargin()`	It gets the margin between the toolbar's border and its buttons
`intgetOrientation()`	It gets the current orientation of the toolbar
`ToolBarUI getUI()`	It gets the toolbar's current user interface
`String getUIClassID()`	It gets the name of the `look and feel` class that draws this component
`boolean isBorderPainted()`	It checks whether the border should be painted
`boolean isFloatable()`	It returns True if the toolbar can be dragged by the user
`boolean isRollover()`	It returns the rollover state
`protected void paintBorder(Graphics g)`	It paints the toolbar's border
`protected String paramString()`	It gets a string representation of this `JToolBar` object

Table 16.16: Methods of the JToolBarclass	
Method	**Does this**
`void setBorderPainted(boolean b)`	It sets whether the border should be painted
`void setFloatable(boolean b)`	It sets whether the toolbar can be made to float
`void setMargin(Insets m)`	It sets the margin between the toolbar's border and its buttons
`void setOrientation(int o)`	It sets the orientation of the toolbar
`void setRollover(boolean rollover)`	It sets the rollover state of this toolbar
`void setUI(ToolBarUIui)`	It sets the `look and feel` object that renders this component
`void updateUI()`	It is called by `UIFactory` when the look and feel has changed

Here's an example in which we add a toolbar with two buttons to a program. Note that in many ways, a toolbar just acts as a component that contains other components —you have to add it to the layout of your program where you want it. However, when you have done that, the user can grasp the toolbar handle, which appears on the left in toolbars, and realign the toolbar against any edge of the window it appears in. In fact, the user can simply leave the toolbar floating in mid space, not aligned against any edge of the window at all.

In the code for this example, we simply use the `JToolBar` class's `add()` method to add two buttons to a toolbar, and we use its `addSeparator()` method to add some space between the buttons. We also add code to make the applet display which of the toolbar buttons the user has clicked, and we add the toolbar to the north section of the content pane's border layout. Here's the code:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
  CODE = ToolBar.class
  WIDTH = 500
  HEIGHT = 280 >
</APPLET>*/
public class ToolBar extends JApplet implements ActionListener{
JButton jbutton1 = new JButton("Button 1", new ImageIcon("bulg2.jpg"));
JButton jbutton2 = new JButton("Button 2", new ImageIcon("bulg2.jpg"));
JButton setrollbutton = new JButton("Set Rollover");
JToolBar jtoolbar = new JToolBar();
JToolBar setrollover = new JToolBar();
public void init(){
Container contentPane = getContentPane();
jbutton1.addActionListener(this);
jbutton2.addActionListener(this);
setrollbutton.addActionListener(this);
jtoolbar.add(jbutton1);
jtoolbar.addSeparator();
jtoolbar.add(jbutton2);
setrollover.add(setrollbutton);
contentPane.add(jtoolbar, BorderLayout.NORTH);
contentPane.add(setrollover, BorderLayout.SOUTH);
jtoolbar.setFloatable(false);
jtoolbar.setRollover(false);}
  public void actionPerformed(ActionEvent e){
  if(jtoolbar.isRollover()){
  if(e.getSource() == jbutton1){
  showStatus("You clicked button 1"); }
  else if (e.getSource() == jbutton2){
```

```
      showStatus("You clicked button 2");}
   }
            else{
            showStatus("Rollover is not set! SET THE ROLLOVERFIRST");}
            if(e.getSource() == setrollbutton){
                           jtoolbar.setRollover(true);
                           showStatus("Setting Rollover Successful");}

      }
   }
```

The result of this code is shown in Figure 16.11, where you can see the two buttons in the toolbar, placed on the top of the screen, and one button in the toolbar, placed at the bottom of the screen:

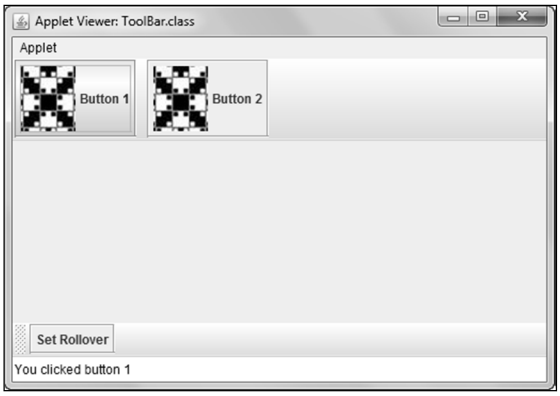


Figure 16.11: Displaying a toolbar with two buttons

As Rollover of toolbar, at the top of the screen, is set to false, you'll not be able to click the buttons to get the button specific message on status bar. To get those messages, first set the Rollover to true by clicking the button, which is in the toolbar placed at the bottom.

Also note the toolbar handle on the left in the toolbar, which the user can use to move the toolbar around, aligning it as needed or leaving it free floating.

Adding Combo Boxes and Other Controls to Toolbars

The Product Support Specialist is in an unhappy mood and says, "Users are complaining again. You have got 20 buttons in your toolbar for selecting the font, and there isn't room for any other buttons." "Hmm," you say, "that could be a problem. I'll put various font options into a combo box in the toolbar instead." The normally lugubrious PSS brightens momentarily.

You can add combo boxes and other controls to toolbars by using the `JToolBar` class's `add()` method. Here's an example in which we add a combo box to the toolbox example developed in the previous solution:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET
   CODE = ToolBr.class
   WIDTH = 500
   HEIGHT = 280 >
</APPLET>*/
public class ToolBr extends JApplet implements ActionListener,
ItemListener{
JButton jbutton1 = new JButton("Button 1", new
```

```
    ImageIcon("bulg2.jpg"));
    JButton jbutton2 = new JButton("Button 2", new
    ImageIcon("bulg2.jpg"));
    JComboBox jcombobox = new JComboBox();
    public void init(){
    Container contentPane = getContentPane();
    JToolBar jtoolbar = new JToolBar();
    jbutton1.addActionListener(this);
    jbutton2.addActionListener(this);
    jcombobox.addItem("Item 1");
    jcombobox.addItem("Item 2");
    jcombobox.addItem("Item 3");
    jcombobox.addItem("Item 4");
    jcombobox.addItemListener(this);
    jtoolbar.add(jbutton1);
    jtoolbar.addSeparator();
    jtoolbar.add(jbutton2);
    jtoolbar.addSeparator();
    jtoolbar.add(jcombobox);
    contentPane.add(jtoolbar, BorderLayout.NORTH);}
      ...
    }
```

When the user selects an item in the combo box, we display that item in the status bar:

```
    public void actionPerformed(ActionEvent e){
            if(e.getSource() == jbutton1) {showStatus("You clicked button 1");}
            else if (e.getSource() == jbutton2) {showStatus("You clicked button 2");}
    }
    public void itemStateChanged(ItemEvent e){
    String outString = "";
    if(e.getStateChange() == ItemEvent.SELECTED)
            outString += "Selected: " + (String)e.getItem();
    else
            outString += "Deselected: " + (String)e.getItem();
    showStatus(outString);
    }
```

The result of this code appears as shown in Figure 16.12:

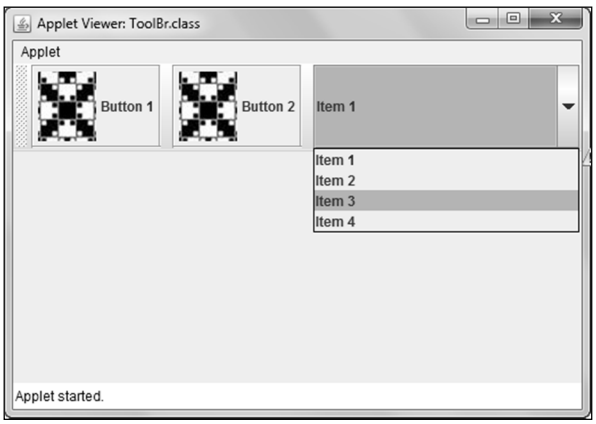


Figure 16.12: Adding a combo box to a toolbar

When the user clicks an item in the combo box, the applet will display that item in the status bar.

Summary

In this chapter, you have read about the menu and toolbars. You have also learned how to create a menu bar and add images to the menu, create toolbars with combo boxes and other controls.

In the next chapter, you'll come across different windows, desktop panes, and dialog boxes.



17

Swing—Windows, Desktop Panes, Inner Frames, and Dialog Boxes

If you need an immediate solution to:	*See page:*
Creating a Window	581
Designing Shaped and Translucent Windows	583
Creating a Frame Window	588
Creating a Desktop Pane	589
Creating Internal Frames	591
Using JOptionPane to Create Dialog Boxes	598
Creating Option Pane Confirmation Dialog Boxes	604
Creating Option Pane Message Dialog Boxes	605
Creating Option Pane Text Field Input Dialog Boxes	606
Creating Option Pane Combo Box Input Dialog Boxes	608
Creating Option Pane Internal Frame Dialog Boxes	609
Creating Dialog Boxes with JDialog	609
Getting Input from Dialog Boxes Created with JDialog	613

In Depth

In this chapter, we'll take a look at handling all kinds of windows in Swing. We'll cover how to work with the `JWindow`, `JFrame`, `JDesktopPane`, `JInternalFrame`, `JOptionPane`, and `JDialog` classes. These classes can be conveniently broken down into two categories—windows and dialog boxes.

Windows

The `JWindow` and `JFrame` classes are the counterparts of the AWT `Window` and `Frame` classes, and they serve the same purpose. You can use the `JWindow` class to create a simple window that's, in fact, nothing more than a blank rectangle. Although, you can add borders and controls to `JWindow` objects, you usually use the `JFrame` class to create windows that you present to the user.

The `JDesktopPane` and `JInternalFrame` classes are new in Swing, although together they represent something that's become common in GUIs—a `multipledocument` interface. Objects of the `JDesktopPane` class present a space in which you can display multiple internal frames of the `JInternalFrame` class. For example, a word processor application may let the user open several views into the same document, or multiple documents, using a desktop with a number of internal frame windows. Internal frames are lightweight, drawn windows that appear inside the other windows. In fact, desktop panes are lightweight components, too, derived from `JLayeredPane`, and you can add internal frame windows to the layers, you select, of a desktop pane.

Dialog Boxes

Swing provides a lot of support for dialog boxes with the `JOptionPane` class. Using the methods of this class, you can display all kinds of dialog boxes—message dialog boxes, confirmation dialog boxes, input dialog boxes, and more. As you'll see here, it's easy to create dialog boxes by using `JOptionPane`.

In addition to `JOptionPane`, Swing also has the `JDialog` class, which you can use as the foundation of your own custom dialog box class. We'll cover how to use both `JOptionPane` and `JDialog` in this chapter in depth.

That's it for the overview of what's in this chapter. As you can see, there's a lot coming up. It's time to turn to the "Immediate Solutions" section, starting with a look at the `JWindow` class.

Immediate Solutions

Creating a Window

The Novice Programmer appears and says, "I want to create an entirely customized window in my program." "For that," you say, "you need the `JWindow` class, which displays an entirely blank window." "OK," says the NP, "can you code it for me?"

The `JWindow` class is much like the AWT `Window` class because it displays a blank window that you can customize as you like. In fact, `JWindow` objects are so unadorned that you'll generally use the `JFrame` class instead. However, if you really need simplicity, `JWindow` is the place to start. Here's the inheritance diagram for this class:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____java.awt.Window
                    |____javax.swing.JWindow
```

You can find the constructors for `JWindow` in Table 17.1 and its methods in Table 17.2:

Table 17.1: Constructors of the JWindow class

Constructor	Does this
`JWindow()`	It constructs a window
`JWindow(Frame owner)`	It constructs a window with the indicated owner frame
`JWindow (GraphicsConfiguration gc)`	It constructs a window with the specified `GraphicsConfiguration` of a screen device
`JWindow(Window owner)`	It constructs a window with the indicated owner window
`JWindow(Window owner, GraphicsConfiguration gc)`	It constructs a window with the specified owner window and `GraphicsConfiguration` of a screen device

Table 17.2: Methods of the JWindow class

Method	Does this
`Graphics getGraphics()`	It creates a graphics context for this component
`protected void addImpl (Component comp, Object constraints, int index)`	It adds components to the content pane instead
`protected JRootPane createRootPane()`	It is called by the constructor methods to create the default root pane
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext` object
`Container getContentPane()`	It gets the `contentPane` object for this window
`Component getGlassPane()`	It gets the `glassPane` object for this window
`JLayeredPane getLayeredPane()`	It gets the `layeredPane` object for this window
`JRootPane getRootPane()`	It gets the `rootPane` object for this window
`TransferHandler getTransferHandler()`	It gets the `transferHandler` property
`protected boolean isRootPaneCheckingEnabled()`	It determines whether calls to `add` and `setLayout` cause an exception to be thrown
`protected String paramString()`	It gets a string representation of this `JWindow`
`void remove(Component comp)`	It removes the indicated component from this container

Table 17.2: Methods of the JWindow class

Method	Does this
`void repaint (long time, int x, int y, int width, int height)`	It repaints the specified rectangle of this component in milliseconds
`void setContentPane (Container contentPane)`	It sets the `contentPane` property
`void setGlassPane (Component glassPane)`	It sets the `glassPane` property
`void setLayeredPane (JLayeredPane layeredPane)`	It sets the `layeredPane` property
`void setLayout (LayoutManager manager)`	By default, the layout of this component may not be set
`protected void setRootPane (JRootPane root)`	It sets the `rootPane` property
`protected void setRootPaneCheckingEnabled (boolean enabled)`	It determines whether calls to `add` and `setLayout` cause an exception to be thrown
`void setTransferHandler(TransferHandler newHandler)`	It sets the transferHandler property, which is a mechanism to support transfer of data into this component
`void update(Graphics g)`	It calls the paint(g) method to draw the graphics
`protected void windowInit()`	It is called by constructors to initialize the `JWindow` properly

When you create a `JWindow` object, you can add a border to it with the root pane's `setBorder()` method, show the window with the `setVisible(true)` method, hide it with the `setVisible(false)` method, and get rid of it with the `dispose()` method. Note that the default layout in a `JWindow` object is a border layout.

We'll create a `JWindow` object now for example. In this case, we add a button to an applet and display the window when that button is clicked. We also add a border to the window and a button that, when clicked, will dispose of the window. Note that `JWindow` objects have content panes just like other windows, so we add the button to that pane. Also note that before a window can be displayed, it must be given a size, which we do here with the `setBounds()` method. Here's the code:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
   CODE = Window.class
   WIDTH = 350
   HEIGHT = 280 >
</APPLET> */
public class Window extends JApplet implements ActionListener {
   JWindow jwindow = new JWindow();
   public void init() {
         Container contentPane = getContentPane();
         JButton jbutton = new JButton("Display window");
         JButton jwindowbutton = new JButton("Close");
         contentPane.setLayout(new FlowLayout());
         contentPane.add(jbutton);
         jwindow.getRootPane().setBorder(
   BorderFactory.createRaisedBevelBorder() );
         Container windowContentPane = jwindow.getContentPane();
         windowContentPane.setLayout(new FlowLayout());
         windowContentPane.add(jwindowbutton);
         jwindowbutton.addActionListener(new ActionListener() {
                 public void actionPerformed(ActionEvent e) {
                 jwindow.dispose(); } });
   jbutton.addActionListener(this); }
   public void actionPerformed(ActionEvent e)  {
```

```
            jwindow.setBounds(200, 200, 100, 100);
            jwindow.setVisible(true); }
    }
```

The result of this code appears as shown in Figure 17.1, where you can see the new window that appears when the user clicks the Display window button:

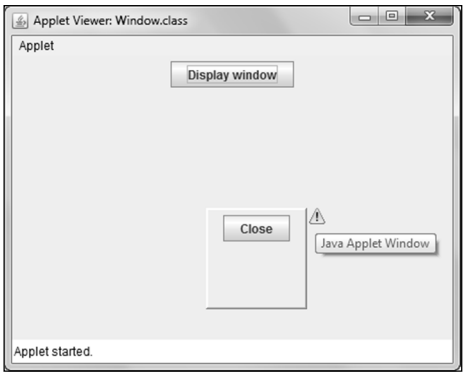


Figure 17.1: A basic JWindow window

As you can see, the window has a border and a button (as well as a warning that Swing adds, indicating that the window is an applet window).

TIP

Although in this example, we pass numeric values to the window's setBounds() method, you can also set the location of the window with respect to its parent window. First, you can get the location of the applet window's content pane like this: Point loc = contentPane.getLocation(). This point is in local coordinates, so you can convert it into screen coordinates, like this: SwingUtilities.convertPointToScreen(loc, contentPane). Then you are free to use loc with setBounds().

Designing Shaped and Translucent Windows

In Java, we can design various shaped and translucent windows. The translucent windows are more attractive to the users, so they are more preferred than the normal windows. Translucency is achieved by changing the opacity of the entire window or by using the per-pixel or uniform translucency that manages the translucency or opacity level of each pixel on the entire window. The translucency value of a window is called alpha value. The general opacity is normally modified compared to per-pixel translucency (opacity) as it consumes less system resources.

We can design different shaped windows in Java by making the desirable area of the windows as transparent. For designing the different shaped and translucent windows, we can use the java.awt.Window class and its subclasses, such as javax.swing.JFrame, javax.swing.JDialog, and java.awt.Frame.

Now, we'll discuss how to create the different shaped and translucent windows in detail.

Creating Translucent Windows

In creating translucent windows, every pixel of the window possesses same translucency. The `setOpacity(float)` method is invoked to maintain the translucency of all the pixels of a window. The argument 'float' passed to this method represents the translucency level of the window, and it ranges between 0 and 1. The lower value represents more transparency of the window.

The code of the TranslucentWindowEg.java source file that uses the setOpacity() method to make 25% translucent window is as follows:

```
    import java.awt.*;
    import javax.swing.*;
    import static java.awt.GraphicsDevice.WindowTranslucency.*;
    public class TranslucentWindowEg extends JFrame {
        public TranslucentWindowEg() {
```

```
            super("This is a Translucent Window");
            setLayout(new GridBagLayout());
            setSize(300,200);
            setLocationRelativeTo(null);
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            //Add a sample button.
            add(new JButton("A Button"));
        }
        public static void main(String[] args) {
            // Determine if the GraphicsDevice supports translucency.
            GraphicsEnvironment ge =
                GraphicsEnvironment.getLocalGraphicsEnvironment();
            GraphicsDevice gd = ge.getDefaultScreenDevice();
            //If translucent windows aren't supported, exit.
            if (!gd.isWindowTranslucencySupported(TRANSLUCENT)) {
                System.err.println(
                    "Translucency is not supported");
                    System.exit(0);
            }
            JFrame.setDefaultLookAndFeelDecorated(true);
            // Create the GUI on the event-dispatching thread
            SwingUtilities.invokeLater(new Runnable() {
                @Override
                public void run() {
                    TranslucentWindowEg tw = new TranslucentWindowEg();
                    // Set the window to 75% opaque (25% translucent).
                    tw.setOpacity(0.75f);
                    tw.setVisible(true);
                }
            });
        }
    }
```

We can see the result of this code in Figure 17.2:

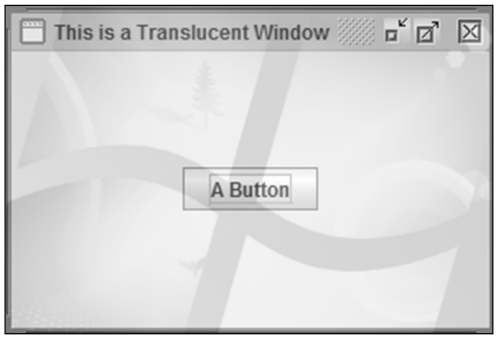


Figure 17.2: A translucent window

You see in the figure that the translucent window displays a button, and the window is 75% opaque.

Per-Pixel Translucency Windows

For creating per-pixel translucent windows, the alpha values need to be defined over the rectangular area on the window. When the alpha value is 0, 255, and 128, the pixel is fully transparent, fully opaque, and 50 percent transparent, respectively.

To create a smooth interpolation between the alpha values, the GradientPaint class is used. By invoking the setBackground(new Color(0,0,0,0)) method on the window, the software to use the alpha values renders per-pixel translucency. In fact, invoking setBackground(new Color(0,0,0,alpha) where alpha is less than 255 creates per-pixel transparency windows. So, if you invoke setBackground(new Color(0,0,0,128)), the window is designed with 50 percent translucency for each background pixel.

The steps required to implement the code using per-pixel translucency are as follows:

1. Invoke setBackground (new Color(0,0,0,0)) on the window.
2. Create a JPanel instance that overrides the paintComponent() method.
3. Create a GradientPaint instance in the paintComponent() method.

   The GradientPaint class smoothly estimates the alpha values from the top to the bottom of the rectangle.
4. Set the GradientPaint instance as the panel's paint() method.

The code for designing per-pixel translucent window is as follows:

```
import java.awt.*;
import javax.swing.*;
import static java.awt.GraphicsDevice.WindowTranslucency.*;
public class PerPixelTranslucentWindowEg extends JFrame {
    public PerPixelTranslucentWindowEg () {
        super("Per-Pixel Translucent Window");
        setBackground(new Color(0,0,0,0));
        setSize(new Dimension(300,200));
        setLocationRelativeTo(null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel panel = new JPanel() {
          @Override
          protected void paintComponent(Graphics g) {
           if (g instanceof Graphics2D) {
            final int R = 240;
            final int G = 240;
           final int B = 240;
                    Paint p =
                        new GradientPaint(0.0f, 0.0f, new Color(R, G, B, 0),
                            0.0f, getHeight(), new Color(R, G, B, 255), true);
                    Graphics2D g2d = (Graphics2D)g;
                    g2d.setPaint(p);
                    g2d.fillRect(0, 0, getWidth(), getHeight());
                }
            }
        };
        setContentPane(panel);
        setLayout(new GridBagLayout());
        add(new JButton("kogent"));
    }
    public static void main(String[] args) {
        // Determine what the GraphicsDevice can support.
        GraphicsEnvironment ge =
            GraphicsEnvironment.getLocalGraphicsEnvironment();
        GraphicsDevice gd = ge.getDefaultScreenDevice();
        boolean isPerPixelTranslucencySupported =
            gd.isWindowTranslucencySupported(PERPIXEL_TRANSLUCENT);
        //If translucent windows aren't supported, exit.
        if (!isPerPixelTranslucencySupported) {
            System.out.println(
                "Per-pixel translucency is not supported");
                System.exit(0);
        }
        JFrame.setDefaultLookAndFeelDecorated(true);
        // Create the GUI on the event-dispatching thread
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                PerPixelTranslucentWindowEg gtw = new
                    PerPixelTranslucentWindowEg ();
                // Display the window.
                gtw.setVisible(true);
            }
```

```
            });
        }
    }
```

The result for the preceding program of per-pixel translucent window is shown in Figure 17.3:

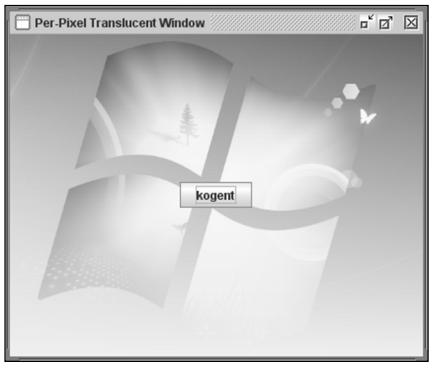


Figure 17.3: Creating per-pixel translucent window

You can see that it is displaying Kogent on the button.

Uniform Translucency Windows

For creating uniform translucency window, translucency should be kept constant for every pixel. This is achieved by invoking the `setOpacity(float)` method in the Window class. The invoked float argument denotes window's transparency level and should possess an inclusive value between 0 and 1. There is also a corresponding `getOpacity()` method.

The code shown below creates a window that is 15 percent opaque (85 percent translucent):

```
    import java.awt.*;
    import javax.swing.*;
    import static java.awt.GraphicsDevice.WindowTranslucency.*;
    public class UniformTranslucentWindowDemo extends JFrame {
    public UniformTranslucentWindowDemo() {
            super("Uniform Translucent Window");
            setLayout(new GridBagLayout());
            setSize(300,200);
            setLocationRelativeTo(null);
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

            //Add a sample button.
            add(new JButton("Welcome Button"));
        }

            public static void main(String[] args) {
        // Determine if the GraphicsDevice supports translucency.
            GraphicsEnvironment ge =
            GraphicsEnvironment.getLocalGraphicsEnvironment();
            GraphicsDevice gd = ge.getDefaultScreenDevice();

            //If translucent windows aren't supported, exit.
            if (!gd.isWindowTranslucencySupported(TRANSLUCENT)) {
             System.err.println(  "Translucency is not supported");
              System.exit(0);
            }
            JFrame.setDefaultLookAndFeelDecorated(true);

            // Create the GUI on the event-dispatching thread
            SwingUtilities.invokeLater(new Runnable() {
                @Override
                public void run() {
                    UniformTranslucentWindowDemo tw = new UniformTranslucentWindowDemo();
```

```
                    // Set the window to 15% opaque (85% translucent).
                    tw.setOpacity(0.15f);

                    // Display the window.
                    tw.setVisible(true);
                }
            });
        }
    }
```

The result is shown in Figure 17.4 showing 15 percent opaqueness on the window:



Figure 17.4: Creating uniform translucent window

Creating Different Shaped Windows

Different shaped windows are created by clipping the window using the `setShape(Shape)` method. Through this method, the different shapes are set to the windows that appear eye-catching to the users. The clipping of the window depends on the passed shape argument. The `setShape()` method is called in the `componentResized()` method of the component event listener where the shape is accurately calculated and set.

Let us write the code for creating the oval-shaped window named as `ShapedWindowEg.java` as follows:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.geom.Ellipse2D;
import static java.awt.GraphicsDevice.WindowTranslucency.*;

public class ShapedWindowEg extends JFrame {
    public ShapedWindowEg () {
        super("ShapedWindow");
        setLayout(new GridBagLayout());

        // It is best practice to set the window's shape in
        // the componentResized method.  Then, if the window
        // changes size, the shape will be correctly recalculated.
        addComponentListener(new ComponentAdapter() {
            // Give the window an elliptical shape.
            // If the window is resized, the shape is recalculated here.
            @Override
            public void componentResized(ComponentEvent e) {
                setShape(new Ellipse2D.Double(0,0,getWidth(),getHeight()));
            }
        });

        setUndecorated(true);
        setSize(300,200);
```

```
            setLocationRelativeTo(null);
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            add(new JButton("I am a Button"));
        }

        public static void main(String[] args) {
            // Determine what the GraphicsDevice can support.
            GraphicsEnvironment ge =
                GraphicsEnvironment.getLocalGraphicsEnvironment();
            GraphicsDevice gd = ge.getDefaultScreenDevice();
            final boolean isTranslucencySupported =
                gd.isWindowTranslucencySupported(TRANSLUCENT);

            //If shaped windows aren't supported, exit.
            if (!gd.isWindowTranslucencySupported(PERPIXEL_TRANSPARENT)) {
                System.err.println("Shaped windows are not supported");
                System.exit(0);
            }

            //If translucent windows aren't supported,
            //create an opaque window.
            if (!isTranslucencySupported) {
                System.out.println(
                    "Translucency is not supported, creating an opaque window");
            }

            // Create the GUI on the event-dispatching thread
            SwingUtilities.invokeLater(new Runnable() {
                @Override
                public void run() {
                    ShapedWindowEg sw = new ShapedWindowEg ();
             // Set the window to 50% translucency, if supported.
                    if (isTranslucencySupported) {
                        sw.setOpacity(0.8f);
                    }
             // Display the window.
                    sw.setVisible(true);
                }
            });
        }
    }
```

The result of the code is shown in Figure 17.5, showing the oval shape of a window:



Figure 17.5: Using the JFrame class

Creating a Frame Window

"So" the NP says, "I was wrong. I didn't want to start with just a basic window; I want to use a frame window that already supports resizing, minimizing, maximizing, and so on." "OK," you say. "It looks like you want the `JFrame` class."

We have been using the `JFrame` class as the foundation of Swing applications, so you are already familiar with this class. We are including it in this chapter on windows and dialog boxes for completeness.

Here's a quick example of using a `JFrame` window in which we add a panel to the window and then draw in the panel and add a text field to it. Because `JFrame` windows are used as the basis of Swing applications. Here's the code:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JFrameDemo extends JFrame {
         jpanel j;
         public JFrameDemo() {
         super("Here is the window");
         Container contentPane = getContentPane();
         j = new jpanel();
         contentPane.add(j); }
         public static void main(String args[]) {
         final JFrame f = new JFrameDemo();
         f.setBounds(100, 100, 300, 300);
         f.setVisible(true);
         f.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
         f.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
         System.exit(0); } });
         }
}
class jpanel extends JPanel {
         JTextField jtextfield = new JTextField("Kogent Solutions!");
         jpanel() {
         setBackground(Color.pink);
         add(jtextfield); }
         public void paintComponent (Graphics g) {
         super.paintComponent(g);
         g.drawString(" Welcome to Kogent Learning Solutions!", 60, 80);
 }
}
```

You can see the result of this code in Figure 17.6:



Figure 17.6: Using the JFrame class

As you see in the figure, the frame window already displays minimize, maximize, and close buttons.

Creating a Desktop Pane

The Big Boss appears and says, "Our Java word processor isn't keeping pace with the competition. They can open five documents at once." "That's nothing," you say. "Using a desktop pane, I can open 10 documents at once and display them all." "You are hired," the BB says. You say, "I already work here."

The Swing desktop pane mimics a desktop and allows you to display multiple objects on it. In the next solution, we'll add inner frame windows to a desktop pane. Here's the inheritance diagram for the `JDesktopPane` class, which the class Swing uses to support desktop panes:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
              |____javax.swing.JComponent
                    |____javax.swing.JLayeredPane
                          |____javax.swing.JDesktopPane
```

You'll find the constructor of the `JDesktopPane` class in Table 17.3 and its methods in Table 17.4:

Table 17.3: The constructor of the JDesktopPane class

Constructor	Does this
`JDesktopPane()`	It constructs a new `JDesktopPane`

Table 17.4: Methods of the JDesktopPane class

Method	Does this
`protected void addImpl(Component comp, Object constraints, int index)`	It adds the specified component to this container at the specified index
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext`
`JInternalFrame[ ] getAllFrames()`	It gets all `JInternalFrames` displayed in the desktop
`JInternalFrame[]getAllFramesInLayer(int layer)`	It gets all `JInternalFrames` displayed in the given layer of the desktop
`DesktopManager getDesktopManager()`	It gets the `DesktopManager` that handles desktop-specific UI actions
`int getDragMode()`	It gets the current "dragging style" used by the desktop pane
`JInternalFrame getSelectedFrame()`	It returns the currently active `JInternalFrame` in this `JDesktopPane` or null if no `JInternalFrame` is currently active
`DesktopPaneUI getUI()`	It gets the `look and feel` object that draws this component.
`String getUIClassID()`	It gets the name of the `look and feel` class that draws this component
`protected String paramString()`	It gets a string representation of this `JDesktopPane`.
`void remove(int index)`	It removes all the components from this pane
`void removeAll()`	It removes all the components from this container
`JInternalFrame selectFrame(boolean forward)`	It selects the next JInternalFrame in this desktop pane
`void setDesktopManager (DesktopManager d)`	It sets the `DesktopManager` that will handle desktop-specific UI actions
`void setDragMode(int dragMode)`	It sets the "dragging style" used by the desktop pane
`void setSelectedFrame (JInternalFrame f)`	It sets the currently active `JInternalFrame` in this `JDesktopPane`
`void setUI (DesktopPaneUI ui)`	It sets the `look and feel` object that draws this component
`void updateUI()`	It is called by `UIManager` when the look and feel has changed

Here's an example in which we just add a desktop pane to an applet's content pane. The code looks like:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
```

```
    CODE = DesktopFrame.class
    WIDTH = 350
    HEIGHT = 280>
</APPLET> */
public class DesktopFrame extends JApplet {
        JDesktopPane jdesktoppane = new JDesktopPane();
        public void init() {
        Container contentPane = getContentPane();
        contentPane.add(jdesktoppane, BorderLayout.CENTER); }
}
```

The result of this code is shown in Figure 17.7:



Figure 17.7: A basic desktop pane

You can see a blank desktop pane in the figure, but it's not much good to anyone—desktop panes are useful as containers, not components, so you need to add something to them. We'll do that in the next solution, where we cover internal frames.

NOTE

You will learn how to use creating scroll panes in Chapter 13 and also how to use using split panes in Chapter 15 of this book.

Creating Internal Frames

The Big Boss isn't impressed with the desktop pane in your program. "What's the point?" the BB asks. "The point is," you say, "you can display windows inside the desktop pane." "Convince me," the BB says. "No problem," you say.

Internal frames, as supported by the `JInternalFrame` class, are lightweight frame windows that appear inside the other windows. This is one of the biggest and most popular of all the Swing classes, and we'll put it to work here. Here's the inheritance diagram for the `JInternalFrame` class:

```
java.lang.Object
  |____java.awt.Component
        |____java.awt.Container
                |____javax.swing.JComponent
                        |____javax.swing.JInternalFrame
```

You'll find the fields of the `JInternalFrame` class in Table 17.5, its constructors in Table 17.6, and its methods in Table 17.7:

Table 17.5: Fields of the JInternalFrame class

Field	Does this
`protected boolean closable`	It indicates whether the frame can be closed
`static String CONTENT_PANE_PROPERTY`	It is the bound property name
`protected JInternalFrame.JDesktopIcon desktopIcon`	It is displayed when the frame is `iconified`
`static String FRAME_ICON_PROPERTY`	It is the `FRAME_ICON_PROPERTY` bound property name
`protected Icon frameIcon`	It is shown in the top-left corner of the frame
`static String GLASS_PANE_PROPERTY`	It is the `GLASS_PANE_PROPERTY` bound property name
`protected boolean iconable`	It returns True if the frame can be minimized and displayed as an icon image
`static String IS_CLOSED_PROPERTY`	It indicates that the frame is closed
`static String IS_ICON_PROPERTY`	It indicates that the frame is `iconified`
`static String IS_MAXIMUM_PROPERTY`	It indicates that the frame is maximized
`static String IS_SELECTED_PROPERTY`	It indicates that this frame has selected status
`protected boolean isClosed`	It return True if the frame has been closed
`protected boolean isIcon`	It returns True if the frame has been `iconified`
`protected boolean isMaximum`	It returns True if the frame has been expanded to its maximum size
`protected Boolean isSelected`	It returns True if the frame is currently selected
`static String LAYERED_PANE_PROPERTY`	It is the `LAYERED_PANE_PROPERTY` bound property name
`protected boolean maximizable`	It returns True if the frame can be expanded to the size of the desktop pane
`static String MENU_BAR_PROPERTY`	It is the bound `MENU_BAR_PROPERTY` property name
`protected boolean resizable`	It returns True if the frame's size can be changed
`static String ROOT_PANE_PROPERTY`	It is the bound `ROOT_PANE_PROPERTY` property name
`protected JRootPane rootPane`	The `JRootPane` instance for this frame
`protected Boolean rootPaneCheckingEnabled`	It returns True when calling the `add` and `setLayout` methods cause an exception to be thrown
`protected String title`	It displayes the title in the frame's title bar
`static String TITLE_PROPERTY`	It is the bound `TITLE_PROPERTY` property name

Table 17.6: Constructors of the JInternalFrame class

Constructor	Does this
`JInternalFrame()`	It constructs a nonresizable, nonclosable, nonmaximizable, and noniconifiable `JinternalFrame`
`JInternalFrame (String title)`	It constructs a nonresizable, nonclosable, nonmaximizable, and noniconifiable `JInternalFrame` with the given title
`JInternalFrame (String title, boolean resizable)`	It constructs a nonclosable, nonmaximizable, and noniconifiable `JInternalFrame` with the given title, which can be resized if `resizable` is True
`JInternalFrame(String title, boolean resizable, boolean closable)`	It constructs a nonmaximizable and noniconifiable `JInternalFrame` with the given title and with the ability to be resized and closed as given

Table 17.6: Constructors of the JInternalFrame class	
Constructor	**Does this**
`JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable)`	It constructs a noniconifiable `JInternalFrame` with the indicated title and with the ability to be resized, closed, and maximized as given
`JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable, boolean iconifiable)`	It constructs a `JInternalFrame` with the indicated title and with the ability to be resized, closed, maximized, and iconified as given

Table 17.7: Methods of the JInternalFrame class	
Method	**Does this**
`protected void addImpl (Component comp, Object constraints, int index)`	It adds child components to the `contentPane`
`void addInternalFrameListener (InternalFrameListener l)`	It adds the given internal frame listener to receive internal frame events from this internal frame
`protected JRootPane createRootPane()`	It is called by the constructor to set up `JRootPane`
`void dispose()`	It disposes of this internal frame
`protected void fireInternalFrameEvent (int id)`	It fires an internal frame event
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext`
`Container getContentPane()`	It gets the `contentPane`
`int getDefaultCloseOperation()`	It gets the default operation that occurs when the user closes this window
`JInternalFrame.JDesktopIcon getDesktopIcon()`	It gets the `JDesktopIcon` used when this `JInternalFrame` is iconified
`JDesktopPane getDesktopPane()`	It searches the hierarchy for a `JDesktop` instance
`Container getFocusCycleRootAncestor()`	It always returns null because `JInternalFrames` must always be roots of a focus traversal cycle
`Component getFocusOwner()`	If this `JInternalFrame` is active, this method returns the child that has focus
`Icon getFrameIcon()`	It gets the image displayed in the title bar of the frame
`Component getGlassPane()`	It gets the `glassPane` object for this frame
`InternalFrameListener[]getInternalFrameListeners()`	It returns an array of all the `InternalFrameListeners` added to this `JInternalFrame` with `addInternalFrameListener`
`JMenuBar getJMenuBar()`	It gets the current `JMenuBar` for this `JInternalFrame` or null if no menu bar has been set
`Cursor getLastCursor()`	It returns the last cursor that was set by the `setCursor()` method
`Component getMostRecentFocusOwner()`	It returns the child component of this `JInternalFrame` that will receive the focus when this `JInternalFrame` is selected
`int getLayer()`	It is a convenience method for getting the layer attribute of this component
`JLayeredPane getLayeredPane()`	It gets the `layeredPane` object for this frame
`JMenuBar getMenuBar()`	It is deprecated and replaced by `getJMenuBar()`
`Rectangle getNormalBounds()`	If the `JInternalFrame` is not in maximized state, this method returns `getBounds()`; otherwise, it returns the bounds that the `JInternalFrame` would be restored to
`JRootPane getRootPane()`	It gets the `rootPane` object for this frame

Table 17.7: Methods of the JInternalFrame class

Method	Does this
`String getTitle()`	It gets the title of the `JInternalFrame`
`InternalFrameUI getUI()`	It gets the `look and feel` object that draws this `JInternalFrame`
`String getUIClassID()`	It gets the name of the `look and feel` class that draws this component
`String getWarningString()`	It gets the warning string that's displayed with this window
`boolean isClosable()`	It gets whether this `JInternalFrame` can be closed by some user action
`boolean isClosed()`	It returns True if this `JInternalFrame` is currently closed
`boolean isIcon()`	It returns True if the `JInternalFrame` is currently iconified
`boolean isIconifiable()`	It returns True if the `JInternalFrame` can be iconified by some user action
`boolean isMaximizable()`	It returns True if the `JInternalFrame` can be maximized by some user action
`boolean isMaximum()`	It returns True if the `JInternalFrame` is currently maximized
`boolean isResizable()`	It returns True if the `JInternalFrame` can be resized by some user action
`protected boolean isRootPaneCheckingEnabled()`	It returns True if calls to `add` and `setLayout` cause an exception to be thrown
`boolean isSelected()`	It returns True if the `JInternalFrame` is the currently selected frame
`void moveToBack()`	It moves this component to position -1 if its parent is a `JLayeredPane`
`void moveToFront()`	It moves this component to position 0 if its parent is a `JLayeredPane`
`void pack()`	It lays out children of this `JInternalFrame` to use their preferred size
`protected void paintComponent(Graphics g)`	It calls the `paint()` method
`protected String paramString()`	It gets a string representation of this `JInternalFrame`
`void remove(Component comp)`	It removes the indicated component
`void removeInternalFrameListener(InternalFrameListener l)`	It removes the indicated internal frame listener
`void reshape(int x, int y, int width, int height)`	It moves and resizes this component
`void restoreSubcomponentFocus()`	This method directs the internal frame to restore focus to the last subcomponent that had focus
`void setClosable(boolean b)`	It determines whether this `JInternalFrame` can be closed by some user action
`void setClosed(boolean b)`	Passing True to this method closes the frame
`void setContentPane (Container c)`	It sets this `JInternalFrame's` content pane
`void setCursor (Cursor cursor)`	It sets cursor image to specified cursor
`void setDefaultCloseOperation (int operation)`	It sets the operation that will occur by default when the user closes this window
`void setDesktopIcon (JInternalFrame.JDesktopIcon d)`	It sets the `JDesktopIcon` associated with this `JInternalFrame`

Table 17.7: Methods of the JInternalFrame class

Method	Does this
`void setFrameIcon (Icon icon)`	It sets an image to be displayed in the title bar of the frame (usually in the top-left corner)
`void setGlassPane (Component glass)`	It sets this `JInternalFrame's glassPane` property
`void setIcon(boolean b)`	It iconifies and deiconifies the frame
`void setIconifiable (boolean b)`	It specifies that the `JInternalFrame` can be made an icon by some user action
`void setJMenuBar (JMenuBar m)`	It sets the `JMenuBar` for this `JInternalFrame`
`void setLayer (Integer layer)`	It sets the layer attribute of this component
`void setLayeredPane (JLayeredPane layered)`	It sets this `JInternalFrame's layeredPane` property
`void setLayout (LayoutManager manager)`	It sets the layout of the `contentPane`
`void setMaximizable (boolean b)`	It specifies that the `JInternalFrame` can be maximized by some user action
`void setMaximum(boolean b)`	It maximizes and restores the frame
`void setMenuBar(JMenuBar m)`	It is deprecated and replaced by `setJMenuBar(JMenuBar m)`
`void setNormalBounds (Rectangle r)`	It sets the normal bounds for this frame (the bounds that the frame will be restored to from its maximized state)
`void setResizable (boolean b)`	It specifies that the `JInternalFrame` can be resized by some user action
`protected void setRootPane (JRootPane root)`	It sets the `rootPane` property
`protected void setRootPaneCheckingEnabled (boolean enabled)`	It determines whether calls to `add` and `setLayout` cause an exception to be thrown
`void setSelected (boolean selected)`	It selects and deselects the `JInternalFrame`
`void setTitle(String title)`	It sets the `JInternalFrame` title
`void setUI (InternalFrameUI ui)`	It sets the UI delegate for this `JInternalFrame`
`void show()`	It sets the visible state of the object
`void show()`	It shows this internal frame and brings it to the front
`void toBack()`	It sends this internal frame to the back
`void toFront()`	It brings this internal frame to the front
`void updateUI()`	It is called by `UIManager` when the look and feel has changed

We'll now put together an example showing how to use internal frames in a desktop pane. We start by adding a desktop pane to an applet, and we add a `JPanel` object as well. We place a button with the caption "New internal frame" in the panel. When the user clicks that button, the applet displays a new internal frame in the desktop pane. Each internal frame will enclose a text area object of the `JTextArea` class (which you'll see in a few chapters—for our purposes here, it acts like the AWT `TextArea` class), thus allowing the user to enter text.

Here's how we set up the applet with a desktop pane and the button 'New internal frame':

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*
<APPLET
CODE = InternalFrame.class
WIDTH = 450
HEIGHT = 300>
```

```
    </APPLET>
    */
    public class InternalFrame extends JApplet implements ActionListener
    {
    JDesktopPane jdesktoppane = new JDesktopPane();
    static int framenumber = 1;
    public void init()
    {
    JPanel jpanel = new JPanel();
    Container contentPane = getContentPane();
    JButton jbutton = new JButton("New internal frame");
    jpanel.add(jbutton);
    contentPane.add(jpanel, BorderLayout.SOUTH);
    contentPane.add(jdesktoppane, BorderLayout.CENTER);
    jbutton.addActionListener(this);}
    }
```

The real action takes place when the user clicks the button to display a new internal frame. We give each internal frame a new caption in its title bar, so we must keep track of the number of internal frames in a new variable.

Next, we create, configure, and add new internal frames to the desktop pane. Here's how we create and configure the internal frames, setting their locations and titles. We enable them so that the user can close, resize, maximize, and minimize these frames, and we add a text area to the frames' content panes. Note, in particular, that internal frames must be made visible with the `setVisible()` method. After the components have been added to the content pane, the `pack()` method is used to resize the frame window to match its contents. Also note that when internal frames are added to desktop panes by using the `add()` method, a layer must be specified to add those frames to. Here's what the code looks like:

```
    public void actionPerformed(ActionEvent event)
    {
    JInternalFrame jinternalframe = new JInternalFrame();
    Container contentPane = jinternalframe.getContentPane();
    jinternalframe.setLocation(20, 20);
    jinternalframe.setTitle("Internal Frame " + framenumber);
    framenumber++;
    jinternalframe.setClosable(true);
    jinternalframe.setResizable(true);
    jinternalframe.setMaximizable(true);
    jinternalframe.setIconifiable(true);
    jinternalframe.setVisible(true);
    contentPane.setLayout(new FlowLayout());
    contentPane.add(new JTextArea(5, 20), "Center");
    jinternalframe.pack();
    jdesktoppane.add(jinternalframe, 5);
    }
```

The result of this code is shown in Figure 17.8:

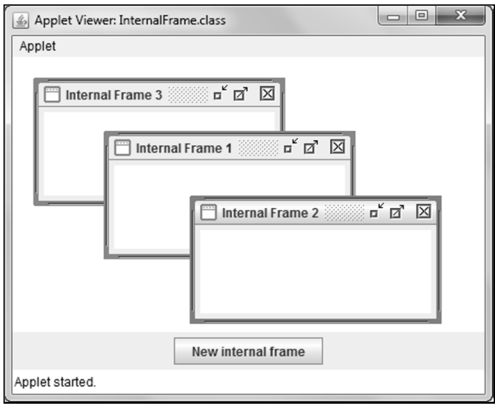


Figure 17.8: Creating internal frames in a desktop pane

As you can see in the figure, you can create multiple internal frame windows in this applet, and you can resize and move them around. You can even minimize them, as you see in the figure.

Now coming to the second example, `InternalframeEg.java`, to create an applet. Following is the code to display the use of Internal Frame in applet:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.net.*;
import java.net.URL;
import javax.swing.border.*;
import javax.swing.event.*;
/* <APPLET
  CODE = InternalframeEg.class
  WIDTH = 400
  HEIGHT = 400>
</APPLET> */
public class InternalframeEg extends JApplet implements ActionListener {
JDesktopPane jdesktoppane = new JDesktopPane();
static int framenumber = 1;
String msg="";
TextArea txt1;
public void init() {
  JPanel jpanel = new JPanel();
  Container contentPane = getContentPane();
  JButton jbutton = new JButton("New internal frame");
  txt1= new TextArea("output displayed here",3,30);
  jpanel.add(jbutton);
  jpanel.add(txt1);
  contentPane.add(jpanel, BorderLayout.SOUTH);
  contentPane.add(jdesktoppane, BorderLayout.CENTER);
  jbutton.addActionListener(this); }
                InternalFrameAdapter activator1 = new InternalFrameAdapter() {
  public void internalFrameActivated(InternalFrameEvent e) {
        JInternalFrame fr = (JInternalFrame)e.getSource();
        InternalFrameListener[] jarr=fr.getInternalFrameListeners();
        txt1.setText(" "+ jarr[1] + "      " + jarr[2]+ "  "); }
public void internalFrameIconified(InternalFrameEvent e) {
                JInternalFrame fr = (JInternalFrame)e.getSource();
                txt1.setText("  " +fr.getMostRecentFocusOwner()); }
public void internalFrameClosed(InternalFrameEvent e) {
                JInternalFrame fr = (JInternalFrame)e.getSource();
                txt1.setText("  " +fr.getFocusCycleRootAncestor()); }
};
InternalFrameAdapter activator2 = new InternalFrameAdapter() {
public void internalFrameOpened(InternalFrameEvent e) {
                JInternalFrame fr = (JInternalFrame)e.getSource();
                txt1.setText("2nd listener event  "); }
};
public void actionPerformed(ActionEvent event) {
        String str= event.getActionCommand();
        if(str.equals("New internal frame")) {
                JInternalFrame jinternalframe = new JInternalFrame();
                Container contentPane = jinternalframe.getContentPane();
                jinternalframe.setLocation(20, 20);
                jinternalframe.setTitle("Frame " + framenumber);
if(framenumber==1) {
                jinternalframe.addInternalFrameListener(activator1);
                jinternalframe.addInternalFrameListener(activator2); }
else if(framenumber==2){ }
framenumber++;
jinternalframe.setClosable(true);
jinternalframe.setResizable(true);
```

```
    jinternalframe.setMaximizable(true);
    jinternalframe.setIconifiable(true);
    jinternalframe.setVisible(true);
    contentPane.setLayout(new FlowLayout());
    contentPane.add(new JTextArea(5, 15), "Center");
    jinternalframe.pack();
    jdesktoppane.add(jinternalframe, 2); }
            }
    }
```

Output of this code is as shown in Figure 17.9:



Figure 17.9: Showing the Initialization of application using Internal Frame

Now click the button `New internal frame` displayed in the applet. This will initialize and create internal frame. The output of this action is shown in Figure 17.10:

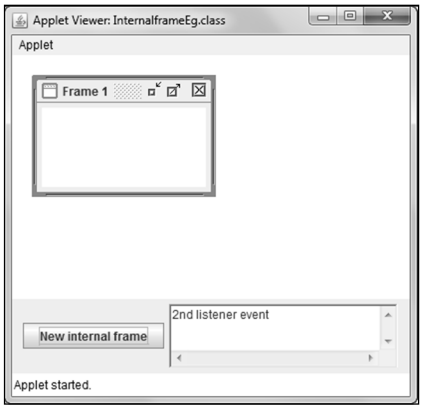


Figure 17.10: Showing the initialization and creation of Internal Frame

Using JOptionPane to Create Dialog Boxes

The Novice Programmer is distraught and says, "The Product Support Specialist wants me to add 12 new dialog boxes to my program! Isn't that a lot of coding?" "Probably not," you say, "using the Swing `JOptionPane` class." "Oh," says the NP, calming down and beginning to smile.

The `JOptionPane` class supports a pane that you use in dialog boxes; in fact, it also provides convenient methods for creating dialog boxes that contain option panes. Here's the inheritance diagram for `JOptionPane`:

```
    java.lang.Object
      |____java.awt.Component
```

```
            |____java.awt.Container
                  |____javax.swing.JComponent
                        |____javax.swing.JOptionPane
```

You'll find the fields of the `JOptionPane` class in Table 17.8, its constructors in Table 17.9, and its methods in Table 17.10:

Table 17.8: Fields of the JOptionPane class

Field	Does this
`static int CANCEL_OPTION`	The return value from the class method if `CANCEL` is chosen
`static int CLOSED_OPTION`	The return value from the class method if the user closes the window without selecting anything
`static int DEFAULT_OPTION`	The look and feel should not supply any options but only use the options from the `JOptionPane`
`static int ERROR_MESSAGE`	It is used for error messages
`protected Icon icon`	The icon used in the pane
`static String ICON_PROPERTY`	The bound property name for the icon
`static int INFORMATION_MESSAGE`	It is used for information messages
`static String INITIAL_SELECTION_VALUE_PROPERTY`	The bound property name for `initialSelectionValue`
`static String INITIAL_VALUE_PROPERTY`	The bound property name for `initialValue`
`protected Object initialSelectionValue`	The initial value to select in `selectionValues`
`protected Object initialValue`	The value that should be initially selected in the options
`static String INPUT_VALUE_PROPERTY`	The bound property name for `inputValue`
`protected Object inputValue`	The value the user has entered
`protected Object message`	The message to display
`static String MESSAGE_PROPERTY`	The bound property name for the message
`static String MESSAGE_TYPE_PROPERTY`	The bound property name for the type
`protected int messageType`	The message type
`static int NO_OPTION`	The return value from the class method if the NO button is clicked
`static int OK_CANCEL_OPTION`	The type used for `showConfirmDialog`
`static int OK_OPTION`	The return value from the class method if OK is chosen
`static String OPTION_TYPE_PROPERTY`	The bound property name for `optionType`
`protected Object[] options`	The options to display to the user
`static String OPTIONS_PROPERTY`	The bound property name for the option
`protected int optionType`	The option type (`DEFAULT_OPTION`, `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION`, or `OK_CANCEL_OPTION`)
`static int PLAIN_MESSAGE`	It indicates that no icon should be used
`static int QUESTION_MESSAGE`	It is used for questions
`static String SELECTION_VALUES_PROPERTY`	The bound property name for `selectionValues`.
`protected Object[] selectionValues`	The array of values, the user can choose from
`static Object UNINITIALIZED_VALUE`	It indicates that the user has not yet selected a value
`protected Object value`	The currently selected value. It will be a valid option, `UNINITIALIZED_VALUE`, or null
`static String VALUE_PROPERTY`	The bound property name for the value

Table 17.8: Fields of the JOptionPane class

Field	Does this
`static String WANTS_INPUT_PROPERTY`	The bound property name for `wantsInput`
`protected Boolean wantsInput`	If True, a `UI` widget will be provided to the user to get input
`static int WARNING_MESSAGE`	It is used for warning messages
`static int YES_NO_CANCEL_OPTION`	The type used for `showConfirmDialog`
`static int YES_NO_OPTION`	The type used for `showConfirmDialog`
`static int YES_OPTION`	The return value from the class method if `YES` is chosen

Table 17.9: Constructors of the JOptionPane class

Constructor	Does this
`JOptionPane()`	It constructs a `JOptionPane` with a test message
`JOptionPane(Object message)`	It constructs an instance of `JOptionPane` to display a message by using the plain-message message type and the default options delivered by the `UI`
`JOptionPane(Object message, int messageType)`	It constructs an instance of `JOptionPane` to display a message with the indicated message type and the default options
`JOptionPane(Object message, int messageType, int optionType)`	It constructs an instance of `JOptionPane` to display a message with the indicated message type and the options
`JOptionPane(Object message, int messageType, int optionType, Icon icon)`	It constructs an instance of `JOptionPane` to display a message with the indicated message type, options, and icon
`JOptionPane(Object message, int messageType, int optionType, Icon icon, Object[] options)`	It constructs an instance of `JOptionPane` to display a message with the indicated message type, icon, and options
`JOptionPane(Object message, int messageType, int optionType, Icon icon, Object[] options, Object initialValue)`	It constructs an instance of `JOptionPane` to display a message with the indicated message type, icon, and options, with the initially selected option indicated

Table 17.10: Methods of the JOptionPane class

Method	Does this
`JDialog createDialog (Component parentComponent, String title)`	It constructs and returns a new `JDialog`
`JDialog createDialog (String title)`	It creates and returns a new parentless `JDialog` with the specified title
`JInternalFrame createInternalFrame(Component parentComponent,String title)`	It constructs and returns an instance of `JInternalFrame`
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext`
`static JDesktopPane getDesktopPaneForComponent (Component parentComponent)`	It gets the indicated component's desktop pane
`static Frame getFrameForComponent (Component parentComponent)`	It gets the indicated component's frame
`Icon getIcon()`	It gets the icon this pane displays
`Object getInitialSelectionValue()`	It gets the initial selection value
`Object getInitialValue()`	It gets the initial value
`Object getInputValue()`	It gets the value the user has entered, if `wantsInput` is True

Table 17.10: Methods of the JOptionPane class

Method	Does this
`int getMaxCharactersPerLineCount()`	It gets the maximum number of characters to place on a line in a message
`Object getMessage()`	It gets the message object which this pane displays
`int getMessageType()`	It gets the message type
`Object[] getOptions()`	It gets the choices the user can make
`int getOptionType()`	It gets the type of options displayed
`static Frame getRootFrame()`	It gets the frame to use for the class methods in which a frame is not provided
`Object[] getSelectionValues()`	It gets the selection values
`OptionPaneUI getUI()`	It gets the `UI` object that implements the look and feel for this component
`String getUIClassID()`	It gets the name of the `UI` class that implements the look and feel for this component
`Object getValue()`	It gets the value the user has selected
`boolean getWantsInput()`	It returns True if a `parentComponent` will be provided for the user to input
`protected String paramString()`	It gets a string representation of this `JOptionPane`
`void selectInitialValue()`	It requests that the initial value be selected, which will set focus to the initial value
`void setIcon(Icon newIcon)`	It sets the icon to display
`void setInitialSelectionValue (Object newValue)`	It sets the initial selection value
`void setInitialValue (Object newInitialValue)`	It sets the initial value that's to be enabled
`void setInputValue (Object newValue)`	It sets the user's input value
`void setMessage (Object newMessage)`	It sets the option pane's message object
`void setMessageType (int newType)`	It sets the option pane's message type
`void setOptions (Object[] newOptions)`	It sets the options, this pane displays
`void setOptionType (int newType)`	It sets the options to display
`static void setRootFrame (Frame newRootFrame)`	It sets the frame to use for class methods in which a frame is not provided
`void setSelectionValues (Object[]newValues)`	It sets the selection values for a pane that provides the user with a list of items to choose from
`void setUI(OptionPaneUI ui)`	It sets the `UI` object that implements the look and feel for this component
`void setValue (Object newValue)`	It sets the value the user has chosen
`void setWantsInput (boolean newValue)`	If `newValue` is true, a component is added to allow the user to input a value
`static int showConfirmDialog (Component parentComponent, Object message)`	It brings up a modal dialog box with the options Yes, No, and Cancel
`static int showConfirmDialog(Component parentComponent, Object message, String title, int optionType)`	It brings up a modal dialog box where the number of choices is determined by the `optionType` parameter

Table 17.10: Methods of the JOptionPane class

Method	Does this
`static int showConfirmDialog(Component parentComponent, Object message, String title, int optionType, int messageType)`	It brings up a modal dialog box where the number of choices is determined by the `optionType` parameter and where the `messageType` parameter determines the icon to display
`static int showConfirmDialog(Component parentComponent, Object message, String title, int optionType, int messageType, Icon icon)`	It brings up a modal dialog box with an indicated icon, where the number of choices is determined by the `optionType` parameter
`static String showInputDialog(Component parentComponent, Object message)`	It shows a question message dialog box requesting input from the user parented to `parentComponent`
`static String showInputDialog(Component parentComponent, Object message, String title, int messageType)`	It shows a dialog box requesting input from the user parented to `parentComponent` with the dialog box having the title `title` and the message type `messageType`
`static Object showInputDialog(Component parentComponent, Object message, String title, int messageType, Icon icon, Object[] selectionValues, Object initialSelectionValue)`	It prompts the user for input in a blocking dialog box where the initial selection, possible selections, and all other options can be indicated
`static String showInputDialog (Object message)`	It shows a question message dialog box requesting input from the user
`static int showInternalConfirmDialog (Component parentComponent, Object message)`	It brings up an internal dialog box with the options Yes, No, and Cancel
`static int showInternalConfirmDialog (Component parentComponent, Object message, String title, int optionType)`	It brings up an internal dialog box where the number of choices is determined by the `optionType` parameter
`static int showInternalConfirmDialog (Component parentComponent, Object message, String title, int optionType, int messageType)`	It brings up an internal dialog box where the number of choices is determined by the `optionType` parameter and where the `messageType` parameter determines the icon to display
`static int showInternalConfirmDialog (Component parentComponent, Object message, String title, int optionType, int messageType, Icon icon)`	It brings up an internal dialog box with an indicated icon, where the number of choices is determined by the `optionType` parameter
`static String showInternalInputDialog (Component parentComponent, Object message)`	It shows an internal question message dialog box requesting input from the user parented to `parentComponent`

Table 17.10: Methods of the JOptionPane class

Method	Does this
`static String showInternalInputDialog (Component parentComponent, Object message, String title, int messageType)`	It shows an internal dialog box requesting input from the user parented to `parentComponent`, with the dialog box having the title `title` and the message type `messageType`
`static Object showInternalInputDialog (Component parentComponent, Object message, String title, int messageType, Icon icon, Object[] selectionValues, Object initialSelectionValue)`	It prompts the user for input in a blocking internal dialog box where the initial selection, possible selections, and all other options can be indicated
`static void showInternalMessageDialog (Component parentComponent, Object message)`	It brings up an internal confirmation dialog box
`static void showInternalMessageDialog (Component parentComponent, Object message, String title, int messageType)`	It brings up an internal dialog box that displays a message using a default icon set by the `messageType` parameter
`static void showInternalMessageDialog (Component parentComponent, Object message, String title, int messageType, Icon icon)`	It brings up an internal dialog box displaying a message and specifying all parameters
`static int showInternalOptionDialog (Component parentComponent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object initialValue)`	It brings up an internal dialog box with an indicated icon, where the initial choice is set by the `initialValue` parameter and the number of choices is determined by the `optionType` parameter
`static void showMessageDialog (Component parentComponent, Object message)`	It brings up a confirmation dialog box
`static void showMessageDialog (Component parentComponent, Object message, String title, int messageType)`	It brings up a dialog box that displays a message using a default icon set by the `messageType` parameter
`static void showMessageDialog (Component parentComponent, Object message, String title, int messageType, Icon icon)`	It brings up a dialog box displaying a message and specifying all parameters
`static int showOptionDialog (Component parentComponent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object initialValue)`	It brings up a modal dialog box with an indicated icon, where the initial choice is determined by the `initialValue` parameter and the number of choices is set by the `optionType` parameter
`void updateUI()`	It is called by `UIManager` when the look and feel has changed

You can use the `JOptionPane` static methods (that is, class methods, which you invoke directly by using the class, like this: `JOptionPane.showMessageDialog()`) to display four types of dialog boxes. Here are those methods and the dialog boxes they create:

- `showMessageDialog()`—It displays some message to the user.
- `showConfirmDialog()`—It asks a confirming question and receives a yes/no/cancel response.
- `showInputDialog()`—It prompts for input.
- `showOptionDialog()`—A configurable dialog box.

For each of these methods, there's also an internal (lightweight) frame version, such as `showInternalMessageDialog()`, `showInternalInputDialog()`, and so on, in case you want to stick to internal frames to conserve system resources.

All these methods can return an integer indicating which button the user has clicked; the possible values are `YES_OPTION`, `NO_OPTION`, `CANCEL_OPTION`, `OK_OPTION`, and `CLOSED_OPTION`. These dialog boxes are *modal*, which means the user must dismiss them before continuing with the rest of the program (clicking any other window in the program just results in a beep).

All this is best understood through examples, so we'll dissect this class by using several examples over the next few solutions.

Creating Option Pane Confirmation Dialog Boxes

The Product Support Specialist says, "We need to make sure the users are okay with this." You ask, "We need to make sure they are okay with displaying the sum after they've clicked the Add button?" "Use a confirmation dialog box," the PSS says.

Confirmation dialog boxes let the user click buttons to confirm or stop an action. You use the `showConfirmDialog()` static method of `JOptionPane` to create this kind of dialog box. The `JOptionPane` constants you can use with this method to indicate which buttons you want displayed are `DEFAULT_ OPTION`, `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION`, and `OK_CANCEL_OPTION`.

Here's an example in which we display a confirmation dialog box with a Yes button and a No button. To display that dialog box, we place a button in an applet, giving the button the caption "Display dialog". When that button is clicked, we use `showConfirmDialog()` to display the new dialog box, and we use the return value from that method to indicate which button was clicked by displaying a message in the applet's status bar. Here's what this looks like in code:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
CODE = DialogConfirm.class
WIDTH = 350
HEIGHT = 280>
</APPLET>
*/
public class DialogConfirm extends JApplet
{
JWindow jwindow = new JWindow();
public void init()
{
final Container contentPane = getContentPane();
JButton jbutton = new JButton("Display dialog");
contentPane.setLayout(new FlowLayout());
contentPane.add(jbutton);
jbutton.addActionListener(new ActionListener()
{
public void actionPerformed(ActionEvent e)
{
int result = JOptionPane.showConfirmDialog((Component)
null, "Choose Yes or No", "Choose Yes or No",
JOptionPane.YES_NO_OPTION);
if (result == JOptionPane.YES_OPTION) {
showStatus("You clicked Yes.");
} else {
```

```
    showStatus("You clicked No.");
    }
    }
    });
    }
    }
```

The result of this code is shown in Figure 17.11, where you can see the Yes/No dialog box:

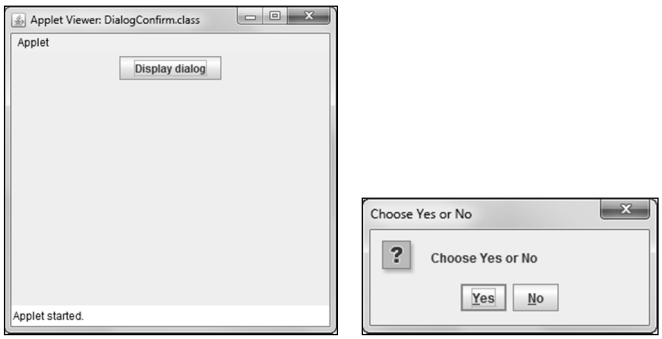


Figure 17.11: A confirmation dialog box

Clicking a button will close the dialog box, and the applet will indicate which button was clicked by displaying a message in the status area.

Creating Option Pane Message Dialog Boxes

"No, no," says the Product Support Specialist. "I didn't mean you should use a confirmation dialog box there—it should be a message dialog box." "OK," you say, "if you are sure."

Message dialog boxes let you display messages to the user, and you use the `JOptionPane` class's `showMessageDialog()` static method to create this kind of dialog box, passing the message you want to display to that method. You can select the type of icon you want to display in these dialog boxes; the `JOptionPane` constants you can use with this method to indicate which icon you want displayed are `INFORMATION_MESSAGE`, `ERROR_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE`, and `PLAIN_MESSAGE`.

Here's an example that shows you how to use all these types of message dialog boxes. In this case, we add a button for each type of dialog box, and clicking each button will display the corresponding dialog box:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

/* <APPLET
   CODE = DialogMessage.class
   WIDTH = 350
   HEIGHT = 280 >
</APPLET> */

public class DialogMessage extends JApplet implements ActionListener {
   JButton jbutton1 = new JButton("Display information dialog");
   JButton jbutton2 = new JButton("Display error dialog");
   JButton jbutton3 = new JButton("Display warning dialog");
   JButton jbutton4 = new JButton("Display question dialog");
   JButton jbutton5 = new JButton("Display plain dialog");

   public void init() {
         Container contentPane = getContentPane();
         contentPane.setLayout(new FlowLayout());
         contentPane.add(jbutton1);
         contentPane.add(jbutton2);
         contentPane.add(jbutton3);
```

```
            contentPane.add(jbutton4);
            contentPane.add(jbutton5);
            jbutton1.addActionListener(this);
            jbutton2.addActionListener(this);
            jbutton3.addActionListener(this);
            jbutton4.addActionListener(this);
            jbutton5.addActionListener(this); }

      public void actionPerformed(ActionEvent e) {
            String dialogtitle = "Dialog";
            String dialogmessage = "Kogent Learning Solutions Inc";
            int dialogtype = JOptionPane.PLAIN_MESSAGE;
            if(e.getSource() == jbutton1) { dialogtype = JOptionPane.INFORMATION_MESSAGE; }
            else if (e.getSource() == jbutton2) {
                    dialogtype = JOptionPane.ERROR_MESSAGE; }
            else if (e.getSource() == jbutton3) {
                    dialogtype = JOptionPane.WARNING_MESSAGE; }
            else if (e.getSource() == jbutton4) {
                    dialogtype = JOptionPane.QUESTION_MESSAGE; }
            else if (e.getSource() == jbutton5) {
                    dialogtype = JOptionPane.PLAIN_MESSAGE; }
            JOptionPane.showMessageDialog((Component) null,
            dialogmessage, dialogtitle, dialogtype); }
   }
```

The result of this code is shown in Figure 17.12, where you can see the buttons in the applet, representing the possible message dialog box types, as well as an information dialog box:

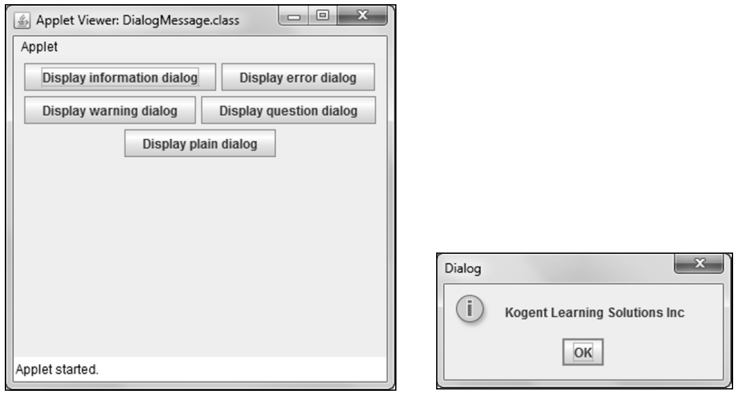


Figure 17.12: Creating different types of message dialog boxes

Creating Option Pane Text Field Input Dialog Boxes

"No," says the Product Support Specialist, "that's not what I wanted at all. I didn't want a message dialog box, I wanted an input dialog box to let the user enter some text." "Hmm," you say.

You can use the `JOptionPane` class's `showInputDialog()` method to display a dialog box with a text field that lets the user enter text. This method will return the text the user enters into the text field or null if the user clicks the Cancel button.

Here's an example in which we just read some text input from the user and display that text in an applet's status bar. The code, itself, is pretty simple. Here it is:

```
   import java.awt.*;
   import javax.swing.*;
   import java.awt.event.*;
   /* <APPLET
     CODE = DialogTextEg.class
     WIDTH = 350
```

```
      HEIGHT = 280>
   </APPLET> */
   public class DialogTextEg extends JApplet implements ActionListener {
      JButton jbutton = new JButton("Display dialog");
      String message = "Enter the text";
            public void init() {
                  Container contentPane = getContentPane();
                  contentPane.setLayout(new FlowLayout());
                  contentPane.add(jbutton);
                  jbutton.addActionListener(this); }
            public void actionPerformed(ActionEvent e) {
                  String result = JOptionPane.showInputDialog(message);
                  if(result == null) { showStatus("You clicked Cancel"); }
                  else { showStatus("You typed: " + result); }
            }
   }
```

The result of this code is shown in Figure 17.13(a). On clicking the Display dialog button, the input dialog box appears, as shown in Figure 17.13(b). When the user clicks the OK button, the dialog box disappears and the text from the text field in the dialog box appears in the applet's status bar as shown in Figure 17.13(c).

Note that we can put `showInputDialog()` to other uses as well—for example, you can display a combo box in a dialog box. See the next solution for the details.



Figure 17.13(a): A basic text-input dialog box

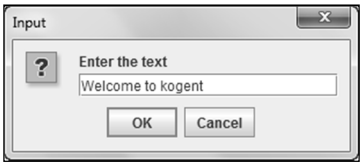


Figure 17.13(b): Enter the text



Figure 17.13(c): Message is displayed

Creating Option Pane Combo Box Input Dialog Boxes

"No, no," the Product Support Specialist says, "I didn't want a text field in that dialog box, I wanted a...." "Combo box?" you ask. "How did you know?" the PSS asks. "Telepathy," you say.

You can use the `JOptionPane` class's `showInputDialog()` method to display a variety of controls; in this solution, we'll use a combo box. To do that, we pass a combo box to `showInputDialog()` as well as an array holding the items we want in that combo box and the item we want selected by default. In this case, we list a variety of desserts and let the user select one. When the user clicks the OK button, we display the selection in the status bar of the applet. Here's the code:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
   CODE = DialogCombo.class
   WIDTH = 350
   HEIGHT = 280>
</APPLET> */
public class DialogCombo extends JApplet implements ActionListener {
   JButton jbutton = new JButton("Display dialog");
   public void init() {
         Container contentPane = getContentPane();
         contentPane.setLayout(new FlowLayout());
         contentPane.add(jbutton);
         jbutton.addActionListener(this); }
   public void actionPerformed(ActionEvent e) {
         String dialogtitle = "Dialog";
         String dialogmessage = "Which one do you like best?";
         String[] desserts = { "cheesecake", "ice cream", "mousse",
                               "carrot cake" };
         String dessert = (String)JOptionPane.showInputDialog(
         DialogCombo.this, dialogmessage, dialogtitle,
         JOptionPane.QUESTION_MESSAGE, null,
         desserts, desserts[0]);
         if(dessert == null) { showStatus("You clicked Cancel"); }
         else { showStatus("Your favorite: " + dessert); }
   }
}
```

The result of this code is shown in Figure 17.14, where you can see the combo box in the dialog box:

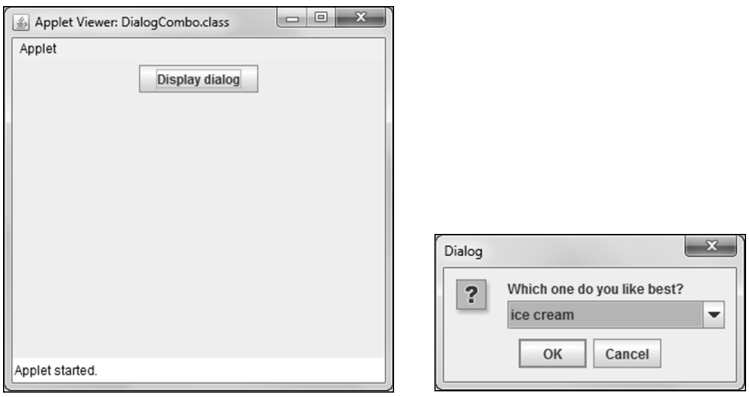


Figure 17.14: An input dialog box with a combo box

When the user selects an item from the combo box and clicks OK, the dialog box closes and the selected item appears in the applet's status bar.

Creating Option Pane Internal Frame Dialog Boxes

In addition to the heavyweight `JOptionPane` dialog boxes created in the previous few examples, you can create lightweight internal dialog boxes that appear inside a container. The `JOptionPane` static methods you can use to create these types of dialog boxes are as follows:

- `showInternalMessageDialog()`—It displays some message to the user
- `showInternalConfirmDialog()`—It asks a confirming question and receives a yes/no/cancel response
- `showInternalInputDialog()`—It prompts for input
- `showInternalOptionDialog()`—A configurable dialog box

Here's an example in which we display an internal message dialog box on a desktop pane when the user clicks the button Display internal dialog. Here's what the code looks like:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
  CODE = OptionPaneInternal.class
  WIDTH = 350
  HEIGHT = 280 >
</APPLET> */
public class OptionPaneInternal extends JApplet implements ActionListener {
  private JButton jbutton = new JButton("Display internal dialog");
  JDesktopPane jdesktoppane = new JDesktopPane();
  public void init() {
        Container contentPane = getContentPane();
        contentPane.add(jbutton, BorderLayout.SOUTH);
        contentPane.add(jdesktoppane);
        jbutton.addActionListener(this); }
  public void actionPerformed(ActionEvent e) {
        JOptionPane.showInternalMessageDialog(
        jdesktoppane, "Kogent Solutions", "Dialog",
        JOptionPane.INFORMATION_MESSAGE); }
}
```

The result appears as shown in Figure 17.15, where you can see the internal message dialog box:

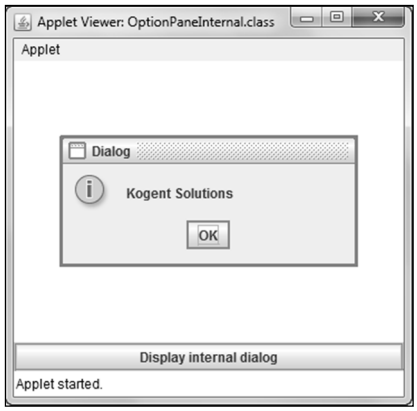


Figure 17.15: An internal message dialog box

Creating Dialog Boxes with JDialog

"Well," says the Novice Programmer, "the simple `JOptionPane` dialog boxes are very nice, but we want to build my own dialog boxes." "From scratch?" you ask. "From scratch," the NP confirms. "Then you want the `JDialog` class," you say.

The `JDialog` class is the foundation of dialog boxes you customize in Swing. Here's the inheritance diagram for this class:

```
java.lang.Object
  |____java.awt.Component
         |____java.awt.Container
                |____java.awt.Window
                       |____java.awt.Dialog
                              |____javax.swing.JDialog
```

You'll find the fields of the `JDialog` class in Table 17.11, its constructors in Table 17.12, and its methods in Table 17.13:

Table 17.11: Fields of the JDialog class

Field	Does this
`protected AccessibleContext accessibleContext`	The accessible context
`protected JRootPane rootPane`	The root pane
`protected Boolean rootPaneCheckingEnabled`	It determines whether root pane checking is enabled

Table 17.12: Constructors of the JDialog class

Constructor	Does this
`JDialog()`	It constructs a modeless dialog box
`JDialog(Dialog owner)`	It constructs a modeless dialog box with the given dialog box as its owner
`JDialog(Dialog owner, boolean modal)`	It constructs a modal or nonmodal dialog box without a title and with the indicated owner dialog box
`JDialog(Dialog owner, String title)`	It constructs a nonmodal dialog box with the indicated title and with the indicated owner dialog box
`JDialog(Dialog owner, String title, boolean modal)`	It constructs a modal or nonmodal dialog box with the indicated title and the indicated owner frame
`JDialog(Dialog owner, String title, boolean modal, GraphicsConfiguration gc)`	It constructs a modal or non-modal dialog with the specified title, owner dialog, and `GraphicsConfiguration`
`JDialog(Frame owner)`	It constructs a nonmodal dialog box without a title with the specified frame as its owner
`JDialog(Frame owner, boolean modal)`	It constructs a modal or nonmodal dialog box without a title and with the indicated owner frame
`JDialog(Frame owner, String title)`	It constructs a nonmodal dialog box with the indicated title and with the indicated owner frame
`JDialog(Frame owner, String title, boolean modal)`	It constructs a modal or nonmodal dialog box with the indicated title and the indicated owner frame
`JDialog(Frame owner, String title, boolean modal, GraphicsConfiguration gc)`	It constructs a modal or nonmodal dialog with the specified title, owner Frame, and `GraphicsConfiguration`
`JDialog(Window owner)`	It constructs a nonmodal dialog box without a title with the specified Window as its owner
`JDialog(Window owner, Dialog.ModalityType modal)`	It constructs a dialog with the specified owner Window, modality, and an empty title
`JDialog(Window owner, String title)`	It constructs a nonmodal dialog with the specified title and owner Window
`JDialog(Window owner, String title, Dialog.ModalityType maodal)`	It constructs a dialog with the specified title, owner Window, and modality

Table 17.13: Methods of the JDialog class

Method	Does this
`protected void addImpl (Component comp,Object constraints, int index)`	They add children to the `contentPane` instead
`protected JRootPane createRootPane()`	It is called by the constructor methods to create the default `rootPane`
`protected void dialogInit()`	It is called by the constructors to initialize the `JDialog` properly
`AccessibleContext getAccessibleContext()`	It gets the `AccessibleContext`
`Container getContentPane()`	It gets the `contentPane`
`int getDefaultCloseOperation()`	It gets the operation that occurs when the user initiates a close operation on this dialog box
`Component getGlassPane()`	It gets the `glassPane` object for this dialog box
`JMenuBar getJMenuBar()`	It gets the menu bar set on this dialog box
`JLayeredPane getLayeredPane()`	It gets the `layeredPane` object for this dialog box
`JRootPane getRootPane()`	It gets the `rootPane` object for this dialog box
`protected Boolean isRootPaneCheckingEnabled()`	It returns True if root pane checking is enabled
`protected String paramString()`	It gets a string representation of this `JDialog`
`protected void processWindowEvent (WindowEvent e)`	It handles window events depending on the state of the `defaultCloseOperation` property
`void remove(Component comp)`	It removes the indicated component from this container
`void repaint(long time, int x, int y, int width, int height)`	It repaints the specified rectangle of this component within time milliseconds
`void setContentPane (Container contentPane)`	It sets the `contentPane` property
`void setDefaultCloseOperation (int operation)`	It sets the operation that will occur by default when the user closes this dialog box
`static void setDefaultLookAndFeelDecorated(boolean dflnf)`	It provides a hint as to whether or not newly created JDialogs should have their Window decorations
`void setGlassPane(Component glassPane)`	It sets the `glassPane` property
`void setJMenuBar (JMenuBar menu)`	It sets the menu bar for this dialog box
`void setLayeredPane (JLayeredPane layeredPane)`	It sets the `layeredPane` property
`void setLayout (LayoutManager manager)`	It sets the layout of the `contentPane` instead
`protected void setRootPane(JRootPane root)`	It sets the `rootPane` property
`void setTransferHandler(TransferHandler newHandler)`	It sets the transferHandler property, which is a mechanism to support transfer of data into this component
`protected void setRootPaneCheckingEnabled (boolean enabled)`	It returns True if calls to `add()` and `setLayout()` will cause an exception to be thrown
`void update(Graphics g)`	It simply calls `paint(g)`

We'll create a dialog box example by using the `JDialog` class here. In this case, we make the dialog box modal by passing a final parameter set to True to the `JDialog` constructor, and we add two buttons to the dialog box—Yes and No. By adding action listeners to these buttons, we can tell which button was clicked, and we can

indicate that in the applet's status bar. After a button is clicked, we call the dialog box's `dispose()` method to dispose of it. Here's what the code looks like (note that as with other Swing windows, you add controls to a `JDialog` object by using its content pane):

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

/*<APPLET
  CODE = Dialog.class
  WIDTH = 350
  HEIGHT = 280>
</APPLET> */

public class Dialog extends JApplet implements ActionListener {
  JLabel jlabel = new JLabel("Click the Yes or No button");
  JButton jbutton1 = new JButton("Display dialog"),
  jbutton2 = new JButton("Yes"),
  jbutton3 = new JButton("No");
  private JDialog dialog = new JDialog((Frame) null, "Dialog", true);

  public void init() {
        Container contentPane = getContentPane();
        Container dialogContentPane = dialog.getContentPane();
        contentPane.setLayout(new FlowLayout());
        contentPane.add(jbutton1);
        dialogContentPane.setLayout(new FlowLayout());
        dialogContentPane.add(jlabel);
        dialogContentPane.add(jbutton2);
        dialogContentPane.add(jbutton3);
        jbutton1.addActionListener(this);
        jbutton2.addActionListener(this);
        jbutton3.addActionListener(this); }

  public void actionPerformed(ActionEvent e) {
        if(e.getSource() == jbutton1) {
                dialog.setBounds(200, 200, 200, 120);
                dialog.setVisible(true); }
        else if(e.getSource() == jbutton2) {
                showStatus("You clicked Yes");
                dialog.dispose(); }
        else if(e.getSource() == jbutton3) {
                showStatus("You clicked No");
                dialog.dispose(); }
  }
}
```

The result of this code is shown in Figure 17.16:



Figure 17.16: A basic JDialog dialog box

As you can see in the figure, the dialog box does display Yes and No buttons. Clicking a button closes the dialog box and a message appears in the applet's status bar indicating which button was clicked.

Note that the dialog box in Figure 17.16 only accepts button input from the user; for a more advanced example, see the next solution.

In the previously mentioned code, you can replace:

```
private JDialog dialog = new JDialog((Frame) null,
"Dialog", true);
```

with:

```
private JDialog dialog = new JDialog((Frame) null,
"Dialog", true, getGraphicsConfiguration());
```

Getting Input from Dialog Boxes Created with JDialog

You can configure dialog boxes created with the `JDialog` class as you like, and we'll take a look at an example in this solution. Here, we display a text field in which the user can enter text. If clicking on the OK button closes the dialog box, the code will display the text the user entered into the text field in the applet's status bar; otherwise, if the user clicks the Cancel button, the code displays the message "You clicked Cancel". Here's what the code looks like:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <APPLET
  CODE = DialogInput.class
  WIDTH = 350
  HEIGHT = 280>
</APPLET> */
public class DialogInput extends JApplet implements ActionListener {
  JLabel jlabel = new JLabel("Enter the text:");
  JTextField jtextfield = new JTextField(15);
  JButton jbutton1 = new JButton("Display dialog"),
  jbutton2 = new JButton("OK"),
  jbutton3 = new JButton("Cancel");
  private JDialog dialog = new JDialog((Frame) null, "Dialog", true);
  public void init() {
        Container contentPane = getContentPane();
        Container dialogContentPane = dialog.getContentPane();
        contentPane.setLayout(new FlowLayout());
        contentPane.add(jbutton1);
        dialogContentPane.setLayout(new FlowLayout());
        dialogContentPane.add(jlabel);
        dialogContentPane.add(jtextfield);
        dialogContentPane.add(jbutton2);
        dialogContentPane.add(jbutton3);
        jbutton1.addActionListener(this);
        jbutton2.addActionListener(this);
        jbutton3.addActionListener(this); }
  public void actionPerformed(ActionEvent e) {
        if(e.getSource() == jbutton1) {
                dialog.setBounds(200, 200, 200, 150);
                dialog.setVisible(true); }
        else if(e.getSource() == jbutton2) {
                showStatus(jtextfield.getText());
                dialog.dispose(); }
        else if(e.getSource() == jbutton3) {
                showStatus("You clicked Cancel");
                dialog.dispose(); }
  }
}
```

The result of this code appears in Figure 17.17, where you can see the dialog box with the text field:

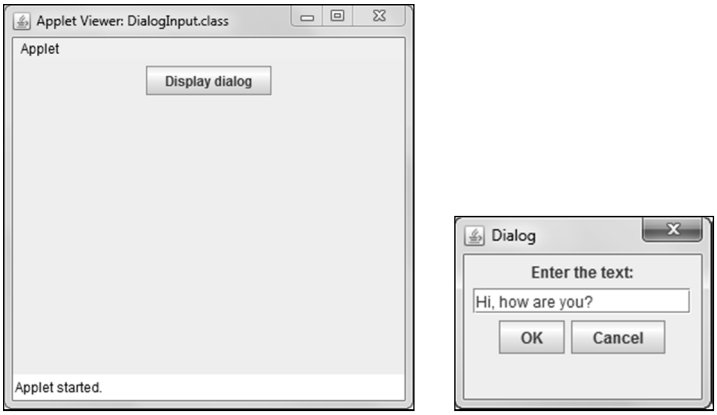


Figure 17.17: A JDialog dialog box with a text field

When the user enters text and clicks OK, the text appears in the status bar of the applet.

Summary

In this chapter, you have learned how to create a simple window and frame-based window. You have also read about working with dialog boxes, such as confirmation dialog box and message dialog box.

In the next chapter, you'll read about the Streams Handling File and I/O.



18

Working with Streams, Files, and I/O Handling

If you need an immediate solution to:	*See page:*
Working with Streams	630
Working with the Reader Class	640
Working with the Writer Class	641
Accepting Input from the Keyboard with the InputStreamReader Class	641
Working with the OutputStreamWriter Class	643
Working with Files	643
Working with the RandomAccessFile Class	649
Working with Character Arrays	651
Working with Buffers	654
Working with the PushbackReader Class	656
Working with the PrintWriter Class	657
Working with the StreamTokenizer Class	659
Implementing the Serializable Interface	661
Working with the Console Class	665
Working with the Clipboard	667
Working with the Printer	667
Printing with the Formatter Class	668
Scanning Input with the Scanner class	671

In Depth

In this chapter, we'll take a look at stream, I/O, and file handling in Java. Using streams is the way you handle I/O in languages such as Java. A stream is composed of bytes or characters (which are used to be considered much the same in Java until Unicode and its 2-byte characters arrived) that you can read from or write to, and you can connect them to various data sources and stores—even byte and character arrays.

Streams, Readers, and Writers

The concept of streams was introduced to make programming more uniform for handling disparate data sources or stores. When using streams, you use the `java.io` package, and you can handle input and output in much the same way—whether you are working with a disk file, a memory buffer, or a network.

In Java, stream I/O is based on the `InputStream`, `OutputStream`, `Reader`, and `Writer` classes. `InputStream` and `OutputStream` are for byte-oriented streams (such as when you work with binary files), and the `Reader` and `Writer` classes are for character-oriented streams (as when you work with text files). You don't use these classes directly; instead, you derive classes from them. In fact, Java has already done a lot of that for you so much, so that there are dozens of classes to work with.

In fact, the I/O stream classes have become somewhat of a thicket in Java—there seems to be at least one available for every possible use and some of you probably haven't thought of. Some streams provide data buffering, some do not. Some are designed to work with data in memory buffers, some with files. We'll cover the important ones here.

Typically, you use a stream class's constructor to create a stream; the stream classes are usually divided into input and output streams, so you select the one that's appropriate. After you have a stream object, you can use its methods—typically `read()` and `write()`—to transfer the data. When you are done with the stream object, you typically use its `close()` method to close it and end your work with the data source or store.

NIO

Initially, New Input Output, also known as NIO, was introduced with the J2SE 1.4 by Sun Microsystems. It is a collection of Java programming language Application Programming Interface (API) that provides features for performing I/O operations.

The APIs of NIO were created for providing accessibility to the low-level I/O operations. The objective of creating such high-level APIs is to enable an implementation that can easily utilize the most effective operations of the existing platform.

The java.nio package and its sub-packages contain the Java NIO APIs. Some of their features are as follows:

- Buffers for the primitive type data
- Character set encoders and decoders
- Pattern matching on the basis of Perl-style regular expressions (in package `java.util.regex`)A new primitive I/O abstraction provisionA file interface inclusion, which supports locking and memory mapping of files up to `Integer.MAX_VALUE` bytes (2 GiB)
- A multiplexed, non-blocking I/O facility for writing scalable servers

Essentials in NIO

NIO is built on two fundamental items—buffers and channels. These are used in data handling—a buffer holds data, while a channel represents an open connection to an I/O device, such as a file or a socket. Generally, to use the NIO system, you obtain a channel to an I/O device and a buffer to hold that data. Operating can be done on the buffer by providing input or output data.

Buffers

Buffers are the integral part of the `java.nio` package. All buffers are subclasses of the `Buffer` class. The `Buffer` class defines the core functionality—current position, limit, and capacity. The current position is the index in the buffer where the next read or write operation takes place. The current position is advanced by most read and write operations. The limit in the index is the end of the buffer. The capacity is the number of elements that the buffer can hold.

The `Buffer` class also supports the `mark()` and `reset()` methods. It defines several methods, which are shown in Table 18.1, and Table 18.2 shows the specific buffer sub classes derived from the `Buffer` class:

Table 18.1: The methods of the Buffer class

Method	Description
`abstract Object array()`	It returns an array backing this buffer.
`abstract int arrayOffset()`	It returns the offset of the first element of the buffer in backing array.
`int capacity()`	It returns the number of elements that the invoking buffer is capable of holding.
`Buffer clear()`	It clears the invoking buffer and returns a reference to the buffer.
`Buffer flip()`	It sets the invoking buffer's limit to the current positions and resets the current positions to 0. It returns a reference to the buffer.
`abstract boolean hasArray()`	It returns true if the buffer is backed by some accessible array.
`boolean hasRemaining()`	It returns true if there are elements remaining in the invoking buffer; otherwise, it returns false.
`abstract boolean isDirect()`	It returns true if the buffer is direct.
`abstract boolean is ReadOnly()`	It returns true if the invoking buffer is read only; otherwise, it returns false.
`int limit()`	It returns the invoking buffer's limit.
`Buffer limit(int n)`	It sets the invoking buffer limit to n. It returns a reference to the buffer.
`Buffer mark()`	It sets the mark and returns a reference to the invoking buffer.
`int position()`	It returns the current positions.
`Buffer position(int n)`	It sets the invoking buffer's current position to n. It returns a reference to the buffer
`int remaining()`	It returns the number of elements between current position and the limit.
`Buffer reset()`	It resets the current position of the invoking buffer to the previously set mark. It returns a reference to the buffer.
`Buffer rewind()`	It sets the positions of the invoking buffer to 0. It returns a reference to the buffer.

Table 18.2: The specific buffer subclasses derived from the Buffer class

`ByteBuffer`	`CharBuffer`
`DoubleBuffer`	`FloatBuffer`
`IntBuffer`	`LongBuffer`
`MappedByteBuffer(a subclass of ByteBuffer; used to map a file to a buffer)`	`ShortBuffer`

All buffers support methods that perform different operations. For example, the `allocate()` method is used for allocating a buffer manually . On the other hand, the `wrap()` method is used for wrapping an array inside a buffer. Moreover, a subsequence of a buffer can be created by using the `slice()` method. The `Buffer` class also supports various `get()` and `put()` methods, which allow you to get data from a buffer or put data into a buffer. The `get()` and `put()` methods are described in Table 18.3:

Table 18.3: The get() and put() methods

Method	Description
`abstract byte get()`	It returns the byte at the current position.
`ByteBuffer get(byte vals[])`	It copies the invoking buffer into the array referred to by vals. It returns a reference to the buffer.
`ByteBuffer get(byte vals[], int start, int num)`	It copies num elements from the invoking buffer into the array referred to by vals, beginning from the index specified by start. It returns a reference to the buffer. If there are not num elements remaining in the buffer, a `BufferUnderflowException` exception is thrown.
`abstract byte get(int idx)`	It returns byte at the index specified by `idx` within the invoking buffer.
`abstract bytebuffer put(byte b)`	It copies b into the invoking buffer at the current position. It returns a reference to the buffer.
`final ByteBuffer put(byte vals[])`	It copies all elements of vals into the invoking buffer, beginning from the current position. It returns a reference to the buffer.
`ByteBuffer put(byte vals[], int start, int num)`	It copies num elements from vals, beginning from start, into the invoking buffer. It returns a reference to the buffer if the buffer cannot hold all the elements a `BufferOverflowException` is thrown.
`ByteBuffer put(ByteBuffer bb)`	It copies the elements in bb to the invoking buffer, beginning from the current position. If the buffer cannot hold all of the elements, a `BufferOverflowException` is thrown. It returns a reference to the buffer
`abstract ByteBuffer put(int idx, byte b)`	It copies b into the invoking buffer at the location specified by `idx`. It returns a reference to the buffer.

Channels

A channel represents an open connection to an I/O source or destination and is defined in the `java.nio.channels package`.

A channel can be retrieved by using the `getChannel()` method on an object supporting channels. Table 18.4 shows its I/O classes:

Table 18.4: The I/O classes supported by the getChannel() method

`DatagramSocket`	`FileInputStream`
`FileOutputStream`	`RandomAccessFile`
`ServerSocket`	`Socket`

To obtain a channel, you first obtain an object of one of these classes and then call `getChannel()` on that object. The type of channel returned totally depends upon the type of object on which the `getChannel()` method is called.

For example, when the `getChannel()` method is called using a `FileInputStream`, `FileOutputStream`, or `RandomAccessFile object`, a channel of type `FileChannel is returned`. When called on a Socket, `getChannel()` method returns a `SocketChannel`.

Channels such as `FileChannel` and `SocketChannel` support various `read()` and `write()` methods that allow you to perform I/O operations through the channel. For example, a few of the `read()` and `write()` methods defined for `FileChannel` are listed in Table 18.5 in which all can throw an `IOException`:

Table 18.5: The read() and write() methods of the FileChannel class

Method	Description
`abstract int read(ByteBuffer bb)`	It reads bytes from the invoking channel into bb until the buffer is full or there is no more input. It returns the number of bytes actually read.
`abstract int read(ByteBuffer bb, long start)`	Beginning at the file location specified by start, read bytes from the invoking channels into bb until the buffer is full or there is no more input. The current position is unchanged. It returns the number of bytes actually read or –1 if start is beyond the end of the file.
`abstract int write(ByteBuffer bb)`	It writes the contents of bb to the invoking channel, starting at the current position. It returns the number of bytes written.
`abstract int write(ByteBuffer bb, long start)`	Beginning at the file location specified by start, writes the contents of bb to the invoking channel. The current position is unchanged. It returns the number of bytes written.

The methods are supported by all channels that let access to and control over them. For example, the `map()` method is supported by `FileChannel` which allows you to map a file to a buffer. Here's a simple example that uses all three types of stream to produce channels that are writeable, readable/writeable, and readable:

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class GetChannelDemo {
private static final int BSIZE = 1024;
public static void main(String[] args) throws Exception
{
FileChannel filechl = new FileOutputStream("file6.txt").getChannel();
filechl.write(ByteBuffer.wrap("Some text ".getBytes()));
filechl.close();
filechl = new RandomAccessFile("file6.txt", "rw").getChannel();
filechl.position(filechl.size());
filechl.write(ByteBuffer.wrap("Some more text".getBytes()));
filechl.close();
filechl = new FileInputStream("file6.txt").getChannel();
ByteBuffer buff = ByteBuffer.allocate(BSIZE);
filechl.read(buff);
buff.flip();
while(buff.hasRemaining())
System.out.print((char)buff.get());}
}
```

Create a file named `file6.txt` before executing this code file in the same location where you are saving Java and the class files. The result is as shown in Figure 18.1:

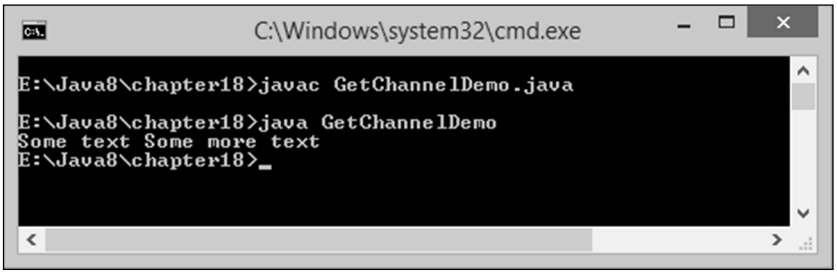


Figure 18.1: The output of the GetChannelDemo class

Charsets and Selectors

NIO supports two more entities—charsets and selectors. A charset allows mapping of bytes to characters. An encoder encodes a sequence of characters into bytes, while a decoder decodes a sequence of bytes into characters. The classes defined in the `java.nio.charset` package support charsets, encoders, and decoders.

Selectors allow you to execute I/O operations through multiple channels. The classes defined in the `java.nio.channels` package support selectors. There are three sets of keys maintained by a selector: key set, selected-key set, and cancelled-key set.

Each time a channel is registered with a selector, a selection key is created. A selection key encapsulates two operation sets—Interest Set and Ready Set.

Enhancements in NIO with Java 8

Several improvements have been made to the standard java.nio.charset.Charset and extended charset implementations. These improvements are as follows:

- Inclusion of a new SelectorProvider implementation for Solaris operating system on the basis of the Solaris event port technique. This implementation is expected to enhance performance or scalability for some server workloads. The /dev/poll SelectorProvider continues to be the default. You can use the Solaris event port technique by running with the system property java.nio.channels.spi.Selector, whose value is set to sun.nio.ch.EventPortSelectorProvider. Reduction in the size of <JDK_HOME>/jre/lib/charsets.jar file
- Performance improvement for the java.lang.String(byte[], *) constructor and the java.lang.String.getBytes() method.

The Path Interface

The Path acts as an object that may be used to locate a file in the file system. Usually, it represents a system-dependent file path. The Path also represents a path that is categorized and composed of an arrangement of directory and file name elements separated by a special separator or delimiter. The root element that identifies a file system hierarchy should be present. The name element that is farthest from the `root` element of the hierarchy is the name of the file or directory and the other name elements are directory names. The Path represents a root and an arrangement of names, or simply one or more name elements. The `Path` is considered to be an empty path if its only name element is empty. Accessing a file using an empty path is similar to accessing the default directory of the file system. The Path interface defines the `getFileName()`, `getParent()`, `getRoot()`, and `subpath()` methods to access the path elements or a subsequence of its name elements.

Paths are utilized with the `Files` class for performing operations on files, directories, and other kinds of files. For example, suppose we want a `BufferedReader` object to read text from a file "file.txt" and the file is located in a directory "info" relative to the current working directory and is UTF-8 encoded. To implement this, the following code snippet should be used:

```
Path path = FileSystems.getDefault().getPath("info", "file.txt");
BufferReader reader = Files.newBufferedReader(path, StandardCharsets.UTF_8);
```

The methods of the `Path` interface class are provided in Table 18.6:

Table 18.6: Methods of the Path interface

Method	Does this
`int compareTo(Path other)`	It compares two abstract paths lexicographically.
`boolean endsWith(Path other)`	It checks whether or not the resulting path ends with the given path.
`boolean endsWith(String other)`	It checks if this path ends with a Path constructed by converting the given path string in exactly the same manner specified by the `endsWith(Path)` method.

Table 18.6: Methods of the Path interface

Method	Does this
`boolean equals(Object other)`	It checks whether or not the given object is a path.
`Path getFileName()`	It returns the name of the file or directory denoted by this path as a `Path` object.
`FileSystem getFileSystem()`	It returns the file system that created this object.
`Path getName(int index)`	It returns a name element of this path as a `Path` object.
`int getNameCount()`	It returns the number of name elements in the path.
`Path getParent()`	It returns the parent path or null if this path does not have a parent.
`Path getRoot()`	It returns the root component of this path as a `Path` object or null if this path does not have a root component.
`int hashCode()`	It computes a hash code for this path.
`boolean isAbsolute()`	It states whether or not this path is absolute.
`Iterator<Path> iterator()`	It returns an iterator over the name elements of this path.
`Path normalize()`	It returns a normalized path in which redundant name of elements are eliminated.
`WatchKey register(WatchService watcher, WatchEvent.Kind<?>... events)`	It registers the file located by this path with a watch service.
`WatchKey register(WatchService watcher, WatchEvent.Kind<?>[] events, WatchEvent.Modifier... modifiers)`	It registers the file located by this path with a watch service. The parameters of this method defines modifications that qualify how the directory is registered.
`Path relativize(Path other)`	It creates a relative path between this `path` and a given path.
`Path resolve(Path other)`	It resolves the given path against this path.
`Path resolve(String other)`	It translates a given path string to a `Path` and resolves it against this Path in exactly the same manner specified by the `resolve` method.
`Path resolveSibling(Path other)`	It resolves the given path against resulting path's parent path.
`Path resolveSibling(String other)`	It translates a given path string to a Path and resolves it against this path's parent path in exactly the same manner specified by the `resolveSibling` method.
`boolean startsWith(Path other)`	It checks if this path starts with the given path.
`boolean startsWith(String other)`	It checks if this path starts with a Path constructed by converting the given path string in exactly the same manner specified by `the startsWith(Path)` method.
`Path subpath(int beginIndex, int endIndex)`	It returns a relative Path that is a subsequence of the name elements of this path.
`Path toAbsolutePath()`	It returns a Path object representing the absolute path of this path.
`File toFile()`	It returns a File object representing this path.
`Path toRealPath(LinkOption... options)`	It returns the real path of an existing file.
`String toString()`	It returns the string representation of this path.
`URI toUri()`	It returns a URI to represent this path.

The **Files** Class

The `Files` class comprises static methods, exclusively, which perform operations on files, directories, or other kinds of files. The inheritance diagram for the `Files` class is as follows:

```
java.lang.Object
  |____java.nio.file.Files
```

The `Files` class exposes the methods described in Table 18.7:

Table 18.7: Methods and their description

Table 18.7: Methods of the Files class

Method	Does this
`static long copy(InputStream in, Path target, CopyOption... options)`	It copies all bytes from an input stream to a file.
`static long copy(Path source, OutputStream out)`	It copies all bytes from a file to an output stream.
`static Path copy(Path source, Path target, CopyOption... options)`	It copies a file to a target file.
`static Path createDirectories(Path dir, FileAttribute<?>... attrs)`	It generates a directory by creating all nonexistent parent directories first.
`static Path createDirectory(Path dir, FileAttribute<?>... attrs)`	It generates a new directory.
`static Path createFile(Path path, FileAttribute<?>... attrs)`	It generates a new and empty file; if the file already exists, then it does not create any file.
`static Path createLink(Path link, Path existing)`	It generates a new link (directory entry) for an existing file (optional operation).
`static Path createSymbolicLink(Path link, Path target, FileAttribute<?>... attrs)`	It generates a symbolic link to a target (optional operation).
`static Path createTempDirectory(Path dir, String prefix, FileAttribute<?>... attrs)`	It generates a new directory in the specified directory using the given prefix to generate its name.
`static Path createTempDirectory(String prefix, FileAttribute<?>... attrs)`	It generates a new directory in the default temporary-file directory and uses the given prefix to generate its name.
`static Path createTempFile(Path dir, String prefix, String suffix, FileAttribute<?>... attrs)`	It generates a new empty file in the specified directory and uses the given prefix and suffix strings to generate its name.
`static Path createTempFile(String prefix, String suffix, FileAttribute<?>... attrs)`	It generates an empty file in the default temporary-file directory using the given prefix and suffix to generate its name.
`static void delete(Path path)`	It deletes a file.
`static boolean deleteIfExists(Path path)`	It deletes a file if it exists.
`static boolean exists(Path path, LinkOption... options)`	It checks whether or not a file exists.
`static Object getAttribute(Path path, String attribute, LinkOption... options)`	It reads the file attribute value.
`static <V extends FileAttributeView> V getFileAttributeView(Path path, Class<V> type, LinkOption... options)`	It returns a file attribute view of a given type.

Table 18.7: Methods of the Files class

Method	Does this
`static FileStore getFileStore(Path path)`	It returns the `FileStore` object specifying the path where the file is stored.
`static FileTime getLastModifiedTime(Path path, LinkOption... options)`	It returns a file's last modified time.
`static UserPrincipal getOwner(Path path, LinkOption... options)`	It returns the owner of a file.
`static Set<PosixFilePermission> getPosixFilePermissions(Path path, LinkOption... options)`	It returns a file's POSIX file permissions.
`static boolean isDirectory(Path path, LinkOption... options)`	It checks whether or not a file is a directory.
`static boolean isExecutable(Path path)`	It checks whether or not a file is executable.
`static boolean isHidden(Path path)`	It states whether or not a file is considered hidden.
`static boolean isReadable(Path path)`	It checks whether or not a file is readable.
`static boolean isRegularFile(Path path, LinkOption... options)`	It checks whether or not a file is a regular file with opaque content.
`static boolean isSameFile(Path path, Path path2)`	It checks if two paths locate the same file.
`static boolean isWritable(Path path)`	It checks whether or not a file is writable.
`static Path move(Path source, Path target, CopyOption... options)`	It moves or renames a file to a target file.
`static BufferedReader newBufferedReader(Path path, Charset cs)`	It opens a file to read text from it and returns a new `BufferedReader` object with default buffer size. This `BufferedReader` object is used to read the content of the file.
`static BufferedWriter newBufferedWriter(Path path, Charset cs, OpenOption... options)`	It opens or creates a file to write text and returns a new `BufferedWriter` object with default buffer size. This `BufferedWriter` object is used to write the text in the file.
`static SeekableByteChannel newByteChannel(Path path, OpenOption... options)`	It opens or creates a file to access it and returns a seekable byte channel, which is used to access the file.
`static SeekableByteChannel newByteChannel(Path path, Set<? extends OpenOption> options, FileAttribute<?>... attrs)`	It opens or generates a file, returning a seekable byte channel to access the file by accepting the path and attributes of the file as arguments.
`static DirectoryStream<Path> newDirectoryStream(Path dir)`	It opens a directory and returns a `DirectoryStream` to iterate over all entries in the directory.
`static DirectoryStream<Path> newDirectoryStream(Path dir, DirectoryStream.Filter<? super Path> filter)`	It opens a directory, returning a `DirectoryStream` to iterate over the entries in the directory. In this method entries are filtered by given filter.
`static DirectoryStream<Path> newDirectoryStream(Path dir, String glob)`	It opens a directory, returning a `DirectoryStream` to iterate over the entries in the directory. In this method entries are filtered by matching the String representation.
`static InputStream newInputStream(Path path, OpenOption... options)`	It opens a file, returning an input stream to read from the file.

Table 18.7: Methods of the Files class

Method	Does this
`static OutputStream newOutputStream(Path path, OpenOption... options)`	It opens or generates a file, returning an output stream that may be used to write bytes to the file.
`static boolean notExists(Path path, LinkOption... options)`	It checks whether or not the file located by this path does not exist.
`static String probeContentType(Path path)`	It probes the content type of a file.
`static byte[] readAllBytes(Path path)`	It reads all the bytes from a file.
`static List<String> readAllLines(Path path, Charset cs)`	It reads all lines from a file.
`static <A extends BasicFileAttributes> A readAttributes(Path path, Class<A> type, LinkOption... options)`	It reads a file's attributes as a bulk operation.
`static Map<String,Object> readAttributes(Path path, String attributes, LinkOption... options)`	It reads a set of file attributes as a bulk operation.
`static Path readSymbolicLink(Path link)`	It reads the target of a symbolic link (optional operation).
`static Path setAttribute(Path path, String attribute, Object value, LinkOption... options)`	It sets the value of a file attribute.
`static Path setLastModifiedTime(Path path, FileTime time)`	It updates a file's last modified time attribute.
`static Path setOwner(Path path, UserPrincipal owner)`	It updates the file owner.
`static Path setPosixFilePermissions(Path path, Set<PosixFilePermission> perms)`	It defines a file's POSIX permissions.
`static long size(Path path)`	It returns the size of a file (in bytes).
`static Path walkFileTree(Path start, FileVisitor<? super Path> visitor)`	It walks a file tree rooted at a given starting file. It does not follow symbolic links automatically but when the option parameter contains the FOLLOW_LINKS option then symbolic links are followed by this method.
`static Path walkFileTree(Path start, Set<FileVisitOption> options, int maxDepth, FileVisitor<? superPath> visitor)`	It walks a file tree rooted at a given starting file. It does not allow to follow symbolic links , and visit all levels of the file tree.
`static Path write(Path path, byte[] bytes, OpenOption... options)`	It writes bytes to a file.
`static Path write(Path path, Iterable<? extends CharSequence> lines, Charset cs, OpenOption... options)`	It writes lines of text to a file.

The **Paths** Class

The `Paths` class includes entirely static methods that return a `Path` by changing a path string or URI. This class is introduced in the Java SE 7 release. It is one of the primary features of the `java.nio.file` package. The inheritance diagram for the `Paths` class is as follows:

```
java.lang.Object
  |____java.nio.file.Paths
```

The methods of the `Paths` class are listed in Table 18.8:

Table 18.8: Methods of the Paths class

Method	Does this
`static Path get(String first, String… more)`	It converts the strings passed as arguments to a Path by joining all the specified strings.
`static Path get(URI uri)`	It changes the given URI to a Path.

The File Attribute Interfaces

An attribute view provides a read-only view of the transparent values, or metadata, associated with objects in a file system. The `FileAttributeView` interface is extended by several other interfaces that view to other specific sets of file attributes. The `FileAttributeView` interface is selected by requesting the `Files.getFileAttributeView(java.nio.file.Path, java.lang.Class<V>, java.nio.file.LinkOption...)` method with a type-token to identify the required view. You can identify views by name as well.

The `FileStoreAttributeView` interface provides access to file store attributes. The `FileStoreAttributeView` interface of a given type is obtained by invoking the `FileStore.getFileStoreAttributeView(java.lang.Class<V>)` method.

Various interfaces of the file attributes are provided in Table 18.9:

Table 18.9: Interface Summary

Interface	Does this
`AclFileAttributeView`	A file attribute view that supports reading or updating a file's ACL or file owner attributes.
`AttributeView`	An object that provides a read-only or updatable view of non-opaque values associated with an object in a file system.
`BasicFileAttributes`	Basic attributes associated with a file in a file system.
`BasicFileAttributeView`	A file attribute view that provides a view of a basic set of file attributes common to many file systems.
`DosFileAttributes`	File attributes associated with a file in a file system that supports legacy "DOS" attributes.
`DosFileAttributeView`	A file attribute view that provides a view of the legacy "DOS" file attributes.
`FileAttribute<T>`	An object that encapsulates the value of a file attribute that can be set atomically when creating a new file or directory by invoking the `createFile()` or `createDirectory()` methods.
`FileAttributeView`	An attribute view that is a read-only or updatable view of non-opaque values associated with a file in a filesystem.
`FileOwnerAttributeView`	A file attribute view that supports reading or updating the owner of a file.
`FileStoreAttributeView`	An attribute view that is a read-only or updatable view of the attributes of a `FileStore`.
`GroupPrincipal`	A `UserPrincipal` representing a group identity used to determine access rights to objects in a file system.
`PosixFileAttributes`	File attributes associated with files on file systems used by operating systems that implement the Portable Operating System Interface (POSIX) family of standards.

Table 18.9: Interface Summary	
Interface	**Does this**
`PosixFileAttributeView`	A file attribute view that provides a view of the file attributes commonly associated with files on file systems used by operating systems that implement the POSIX family of standards.
`UserDefinedFileAttributeView`	A file attribute view that provides a view of a file's user-defined attributes, sometimes known as extended attributes.
`UserPrincipal`	A Principal representing an identity used to determine access rights to objects in a file system.

The *FileSystem* Class

The `FileSystem` class is the collection of objects providing an interface, which is used to access files and other objects present in the file system. It is the default file system used with the `FileSystems.getDefault()` method invoked. This method allows accessibility to the file system that can be easily retrieved by the JVM.

The `FileSystem` class defines methods for creating file systems that provide access to other types of file systems.

The `FileSystem` class provides the following types of objects:

- The `getPath()` method converts the path of a system, which is in String format, into a Path object. This object is used to locate a file and retrieve its content.
- The `getPathMatcher()` method creates a `PathMatcher` object, which is used to perform matching operations on paths.
- The `getFileStores()` method returns an object for performing iteration on the exisiting file stores.
- The `getUserPrincipalLookupService()` method returns the `userPrincipaLookupService`, which is used to lookup users or groups by name.
- The `newWatchService()` method constructs a `WatchService` used to look the objects for changes and events.
- The `getRootDirectory()` method returns an object for performing iteration on the root directories present in the file system.

The file system sometimes have single hierarchy of files with one top-level root directory or several distinct hierarchy of files, each with its own top-level root directory.

Generally, a file system contains one or more essential file stores that are used to store the files on the system. The file attributes or metadata and features of one file store will differ from the other.

When you create an object of the `FileSystem` class, the file system is opened to access files and can be closed by calling its `close()` method. If the file system is closed, then any attempt to access objects from the files will result into the `ClosedFileSystemException` exception. You cannot close the File systems that are created by the default provider. Whenever you want to close the file system, the `close()` method can be invoked but the way a file system will be closed is provider specific. In other words, the type of provider used with the file system will determine during the runtime how the system will be closed.

A file system provides read-only or read-write operations to retrieve the data from the file system. When the file system is created, you can check whether it provides read-only access by using the `isReadOnly()` method of the file system. When you try to write or change in any file store through an object related to its read only file system, the object throws the `ReadOnlyFileSystemException` exception. File systems are secure when they are used by several simultaneous threads. The inheritance diagram for this class is as follows:

```
java.lang.Object
  |____java.nio.file.FileSystem
```

The constructors of the `FileSystem` class are provided in Table 18.10 and its methods in Table 18.11:

Table 18.10: The constructor of the FileSystem class

Constructor	Does this
`Protected FileSystem()`	It is used to initialize a new instance of the `FileSystem` class.

Table 18.11: Methods of the FileSystem class

Method	Does this
`abstract void close()`	It closes the file system that is currently in use.
`abstract Iterable<FileStore> getFileStores()`	It returns an object to iterate over the underlying file stores.
`abstract Path getPath(String first, String... more)`	It changes a `path` string, or a sequence of strings to a Path. The sequence of strings are joined together to form a Path string.
`abstract PathMatcher getPathMatcher(String syntaxAndPattern)`	It returns a `PathMatcher` to perform the math operation on the String representation of Path object.
`abstract Iterable<Path> getRootDirectories()`	It returns an object to iterate over the paths of the root directories.
`abstract String getSeparator()`	It returns the name separator, represented as a string.
`abstract UserPrincipalLookupService getUserPrincipalLookupService()`	It returns the `UserPrincipalLookupService` for this file system (optional operation).
`abstract boolean isOpen()`	It returns a boolean value specifying whether or not this file system is open.
`abstract boolean isReadOnly()`	It returns a boolean value specifying whether or not this file system allows only read-only access to its file stores.
`abstract WatchService newWatchService()`	It creates a new `WatchService` (optional operation).
`abstract FileSystemProvider provider()`	It returns the provider that created this file system.
`abstract Set<String> supportedFileAttributeViews()`	It returns the set of the names of the file attribute views supported by this `FileSystem`.

The *FileSystems* Class

The `FileSystems` class gives collection of methods for file systems. The `getDefault()` method provided by this class returns the default file system and the collection of methods to create other types of file systems.

Whenever the default provider is loaded, the method of this class will definitely be invoked for the first time. The default provider is denoted by the URI scheme "file", which creates the file system that allows you to retrieve the data. But if the process of loading or instantiating the default provider is not successful, an undetermined error occurs.

The `installedProviders()` method is invoked for the first time by calling any of the `newFileSystem()` methods defined by the `FileSystems` class. This method identifies and loads all installed file system providers. With the help of the service-provider loading capability, the installed providers can be loaded, which are defined by service loader class. If the system class loader is not found, then the extension class loader is used. But if there is no extension class loader, then the bootstrap class loader is used for loading the file system.

Generally, you can install the providers by storing them in a JAR file, which is configured on the application classpath. You can also store the providers to be installed in the extension directory. The JAR file includes a provider-configuration, that is, `java.nio.file.spi.FileSystemProvider`, in the source directory META-INF/services. The ordering in which the installed providers are available depends upon the implementation of the provider. If during the instantiation of a provider, the `getScheme()` method returns the same URI pattern

as that of the earlier instantiated provider, then the file system provider is rejected because of duplicity. You should note that the URI schemes are case-insensitive. During instantiation, a provider can retrieve the data related to the default provider from files in the secured manner, but the circular loading of other installed providers should be avoided. (This is because the circular loading of installed providers can cause an unknown error).

The `FileSystems` class also defines some methods that allow you to specify the `ClassLoader` to be used while locating a provider. If a thread starts loading of installed file system providers and another thread requests a method that also wants to load the providers, then the method will block until the loading completes. The inheritance diagram for this class is as follows:

```
java.lang.Object
  |____java.nio.file.FileSystems
```

The methods of the `FileSystems` class are listed in Table 18.12:

Table 18.12: Methods of the FileSystems class

Method	Does this
`static FileSystem getDefault()`	It returns the default `FileSystem`.
`static FileSystem getFileSystem(URI uri)`	It returns a reference to an existing `FileSystem`.
`static FileSystem newFileSystem(Path path, ClassLoader loader)`	It creates a new `FileSystem` to access the contents of a file as a file system.
`static FileSystem newFileSystem(URI uri, Map<String,?> env)`	It creates a new file system that is identified by a URI.
`static FileSystem newFileSystem(URI uri, Map<String,?> env, ClassLoader loader)`	It creates a new file system that is identified by a URI.

The *FileStore* Class

(The `FileStore` class defines the storage for files). A file store is a volume or partition in which files are stored. You can obtain the location of the storage of files by calling the `getFileStore()` method. Another methods of the `FileStore` class, `getFileStores()` method, is used to count the total number of file stores. A file store can support more than one `FileStoreAttributeView` class which provides the read-only or updatable view of the `FileStore` attributes. The inheritance diagram for this class is as follows:

```
java.lang.Object
  |____java.nio.file.FileStore
```

The constructor of the `FileStore` class listed in Table 18.13 and its methods in Table 18.14:

Table 18.13: The constructor of the FileStore class

Constructor	Does this
`protected FileStore()`	It initializes new instance of this class.

Table 18.14: Methods of the FileStore class

Method	Does this
`abstract Object getAttribute(String attribute)`	It reads the value of a file store attribute.
`abstract <V extends FileStoreAttributeView>V getFileStoreAttributeView(Class<V> type)`	It returns a `FileStoreAttributeView` of the given type.
`abstract long getTotalSpace()`	It returns the size of the file store in bytes.
`abstract long getUnallocatedSpace()`	It returns the number of unallocated bytes in the file store.

Table 18.14: Methods of the FileStore class	
Method	**Does this**
`abstract long getUsableSpace()`	It returns the number of bytes available to this Java virtual machine on the file store.
`abstract boolean isReadOnly()`	It states whether this file store is read-only.
`abstract String name()`	It returns the name of this file store.
`abstract boolean supportsFileAttributeView(Class<? extends FileAttributeView> type)`	It states whether or not this file store supports the file attributes identified by the given file attribute view.
`abstract boolean supportsFileAttributeView(String name)`	It states whether or not this file store supports the file attributes identified by the given file attribute view.
`abstract String type()`	It returns the type of this file store.

Prospects of NIO

Only time and usage can tell whether the new approach will someday replace the traditional approach. However, it is certain that the NIO APIs offer an exciting new way to think about and handle various types of operations.

So what more does NIO give us? A lot, but not everything—native code is still required. Some things that could be done only with native code are now provided in NIO.

There's a lot coming up next: I/O is a huge topic in Java. It's time to turn to the "Immediate Solutions" section.

Immediate Solutions

Working with Streams

Generally, programs use the data from external sources, which can be from the remote machines or from the local files on the same system. Java uses the concept of streams for reading data from the system and writing data to it.

Stream sources include files, buffers, sockets, threads, and other streams. Stream destinations include the same objects as stream sources and others as well, such as printers. There are generally two types of streams: input stream and output stream. In case of input stream, a stream of data items flows from a source, whereas, in case of output stream, the data items flow to a destination.

The *InputStream* Class

The Novice Programmer appears and says, "I'm ready to start working with files. Where do I start?" "You start," you say, "with the `InputStream` and `OutputStream` classes."

The `InputStream` and `OutputStream` classes are the base classes for byte-oriented I/O in Java, so it's worth taking a look at the methods that these classes provide to all other byte-oriented stream classes. Here's the inheritance diagram for the `InputStream` class, on which you base input streams:

```
java.lang.Object
  |____java.io.InputStream
```

The `InputStream` is an abstract class that defines Java's model of streaming byte input. All the methods of this class will throw an `IOException`.

You'll find the constructor of the `InputStream` class in Table 18.15 and its methods in Table 18.16:

Table 18.15: The constructor of the InputStream class

Constructor	Does this
`InputStream()`	It constructs an `InputStream` object

Table 18.16: Methods of the InputStream class

Method	Does this
`int available()`	It gets the number of bytes that can be read from this input stream
`void close()`	It closes this input stream
`void mark(int readlimit)`	It marks the current position in this input stream
`boolean markSupported()`	It returns True if this input stream supports the `mark()` and `reset()` methods
`abstract int read()`	It reads the next byte of data from the input stream
`int read(byte[] b)`	It reads some number of bytes from the input stream and stores them into the buffer array `b`
`int read(byte[ ] b, int off, int len)`	It reads up to `len` bytes of data from the input stream into an array of bytes
`void reset()`	It repositions this stream to the position where the `mark()` method was last called on this input stream
`long skip(long n)`	It skips over `n` bytes of data from this input stream

We'll take a look at the `OutputStream` class in the next solution.

The *OutputStream* Class

The counterpart of the `InputStream` class, introduced in the previous section, is the `OutputStream` class, on which you base output streams. The inheritance diagram for the `OutputStream` class is as follows:

```
java.lang.Object
  |____java.io.OutputStream
```

The `OutputStream` class refers to an abstract class defining streaming byte output. All the methods of this class return void and throw `IOException` in case of errors. While `InputStream` implements the `Closeable` interface, `OutputStream` implements the `Closeable` and `Flushable` interfaces both.

You'll find the constructor of the `OutputStream` class in Table 18.17 and its methods in Table 18.18:

Table 18.17: The constructor of the OutputStream class

Constructor	Does this
`OutputStream()`	It constructs an `OutputStream` object.

Table 18.18: Methods of the OutputStream class

Method	Does this
`void close()`	It closes this output stream.
`void flush()`	It flushes this output stream and writes any waiting buffered output bytes.
`void write(byte[ ] b)`	It writes `b.length` bytes from the given byte array to this output stream.
`void write(byte[] b, int off, int len)`	It writes `len` bytes from the given byte array, starting at offset `off`, to this output stream.
`abstract void write(int b)`	It writes the given byte to this output stream.

Now that you have seen `InputStream` and `OutputStream`, it's time to put these classes to work, starting in the next solution.

The *ByteArrayInputStream* Class

"I have got a huge array of bytes in my program," the NP says, "and I'm always losing my place in it. Too bad you can't treat an array of bytes like a file and open an input stream from it!" "But you can," you say. The NP is astounded.

You can use the `ByteArrayInputStream` class to open an input stream from an array of bytes in memory. Here's the inheritance diagram for this class:

```
java.lang.Object
  |____java.io.InputStream
        |____java.io.ByteArrayInputStream
```

You'll find the fields of the `ByteArrayInputStream` class in Table 18.19, its constructors in Table 18.20, and its methods in Table 18.21:

Table 18.19: Fields of the ByteArrayInputStream class

Field	Does this
`protected byte[] buf`	It is an array of bytes that was supplied by the creator of the stream.
`protected int count`	It is the number of characters in the input stream buffer.
`protected int mark`	It is the currently marked position in the stream.
`protected int pos`	It is the index of the next character to read from the input stream.

Table 18.20: Constructors of the ByteArrayInputStream class	
Constructor	**Does this**
`ByteArrayInputStream(byte[] buf)`	It constructs a `ByteArrayInputStream` object.
`ByteArrayInputStream(byte[] buf, int offset, int length)`	It constructs a `ByteArrayInputStream` object that uses `buf` as its buffer array. In this case, we are just creating a memory buffer with the text.

Table 18.21: Methods of the ByteArrayInputStream class	
Method	**Does this**
`int available()`	It gets the number of bytes that can be read from this input.
`void close()`	It closes this input stream.
`void mark(int readAheadLimit)`	It sets the current marked position in the stream.
`boolean markSupported()`	It returns True if this `ByteArrayInputStream` object supports `mark()/reset()`.
`int read()`	It reads the next byte of data.
`int read(byte[] b, int off, int len)`	It reads up to `len` bytes of data into an array of bytes from this input stream.
`void reset()`	It resets the buffer to the marked position.
`long skip(long n)`	It skips `n` bytes of input from this input stream.

Here's an example. In this case, we are just creating a memory buffer with the text "Dreamtech Press" in it and then opening a `ByteArrayInputStream` object to that buffer, reading and displaying the text byte by byte:

```
import java.io.*;
class ByteArrayInputStreamDemo
{
  public static void main(String args[]) throws IOException
  {
        byte data[] = "Dreamtech Press".getBytes();
        ByteArrayInputStream in = new ByteArrayInputStream(data);
        int character;
        while((character = in.read()) != -1) {
                System.out.print((char) character); }
  }
}
```

The output from this code is shown in Figure 18.2, showing a string stored in an array of bytes:

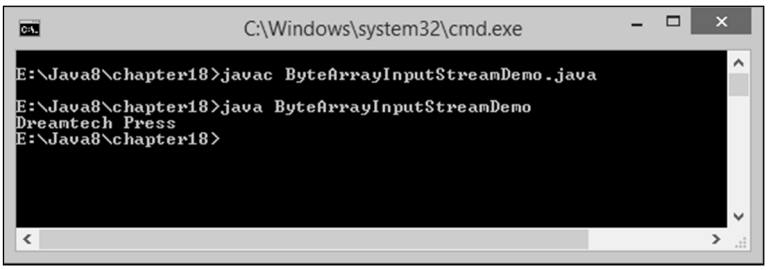


Figure 18.2: An array storing a number of bytes

That's all it takes.

Take the case of another example, where the `reset()` method is used to read the same input twice, i.e. the string "Dreamtech" is read and printed once in lowercase and then again in uppercase:

```
import java.io.*;
class ResetDemo
{
  public static void main(String args[]) throws IOException
  {
        String p = "Dreamtech";
        byte B[] = p.getBytes();
        ByteArrayInputStream in = new ByteArrayInputStream(B);
        for (int j=0; j<2; j++)
        {
                int N;
                while ((N = in.read()) != -1)
                {
                        if (j == 0) { System.out.print((char) N); }
                        else {
                        System.out.print(Character.toUpperCase((char) N)); }
                }
                System.out.println();
                in.reset(); }
  }
}
```

The output is shown in Figure 18.3, showing the same input printed twice:

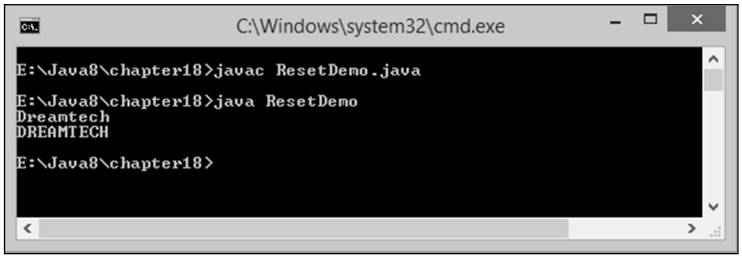


Figure 18.3: Input printed twice

The *ByteArrayOutputStream* Class

"So you can read bytes with `ByteArrayInputStream`," says the Novice Programmer. "Can you also write them?" "Sure," you say, "with `ByteArrayOutputStream`."

The `ByteArrayOutputStream` class lets you write streams of bytes to buffers. Here's the inheritance diagram for this class:

```
java.lang.Object
  |____java.io.OutputStream
        |____java.io.ByteArrayOutputStream
```

You'll find the fields of the `ByteArrayOutputStream` class in Table 18.22, its constructors in Table 18.23, and its methods in Table 18.24:

Table 18.22: Fields of the ByteArrayOutputStream class

Field	Does this
`protected byte[] buf`	The buffer where data is stored
`protected int count`	The number of bytes in the buffer

Table 18.23: Constructors of the ByteArrayOutputStream class

Constructor	Does this
`ByteArrayOutputStream()`	It constructs a new byte array output stream
`ByteArrayOutputStream(int size)`	It constructs a new byte array output stream with a buffer of the given size

Table 18.24: Methods of the ByteArrayOutputStream class	
Method	**Does this**
`void close()`	It closes this output stream
`void reset()`	It resets the count field of this byte array output stream to zero
`int size()`	It gets the current size of the buffer
`byte[] toByteArray()`	It constructs a newly allocated byte array
`String toString()`	It converts the buffer's contents into a string
`String toString(int hibyte)`	It is deprecated and it uses the `toString()` method
`String toString(String enc)`	It converts the buffer's contents into a string
`void write(byte[ ] b, int off, int len)`	It writes `len` bytes from the given byte array starting at offset `off`
`void write(int b)`	It writes the given byte to this byte array output stream
`void writeTo(OutputStream out)`	It writes the complete contents of this byte array output stream to the given output stream argument

Here's an example. In this case, we are writing bytes to a `ByteArrayOutputStream` object, writing bytes to a memory buffer, and sending bytes to a file output stream:

```
import java.io.*;
class ByteArrayOutputStreamDemo
{
  public static void main(String args[]) throws IOException
  {
        ByteArrayOutputStream bytearrayoutputstream = new
        ByteArrayOutputStream();
        byte data[] = " Welcome to Dreamtech Press ".getBytes();
        bytearrayoutputstream.write(data);
        System.out.println(bytearrayoutputstream.toString());
        byte buffer[] = bytearrayoutputstream.toByteArray();
        for (int loop_index = 0; loop_index < data.length; loop_index++)
        {
                System.out.print((char) buffer[loop_index]); }
                OutputStream fileoutputstream = new FileOutputStream("file.txt");
                bytearrayoutputstream.writeTo(fileoutputstream);
                fileoutputstream.close(); }
}
```

The result can be seen in Figure 18.4, showing the bytes to be stored in the buffer:

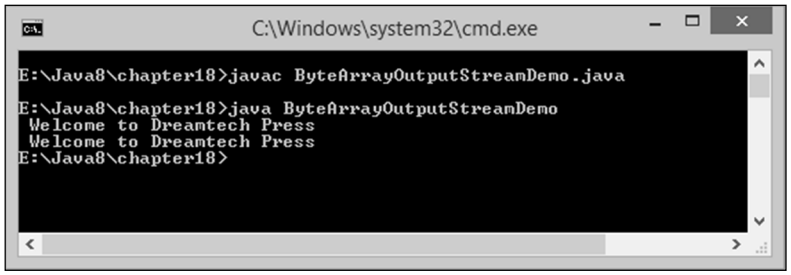


Figure 18.4: Bytes stored in the buffer

The *BufferedInputStream* Class

"Good Lord!," says the Novice Programmer, "My code interprets data from an input stream, and sometimes it'd be handy to be able to reset the stream and go back to an earlier position." "You can do that with buffered streams," you say. "Wow!" the NP says.

Using the `BufferedInputStream` class, you can use the `mark()` and `reset()` methods to actually move backward in a buffered input stream. The inheritance diagram for this class is as follows:

```
java.lang.Object
   |____java.io.InputStream
         |____java.io.FilterInputStream
               |____java.io.BufferedInputStream
```

You'll find the fields of the `BufferedInputStream` class in Table 18.25, its constructors in Table 18.26, and its methods in Table 18.27:

Table 18.25: Fields of the BufferedInputStream class

Field	Does this
`protected byte[] buf`	The internal buffer where the data is stored.
`protected int count`	The count of bytes in the buffer.
`protected int marklimit`	The maximum read-ahead allowed after a call to the `mark()` method.
`protected int markpos`	The value of the `pos` field at the time the last `mark()` method was called.
`protected int pos`	The position in the buffer.

Table 18.26: Constructors of the BufferedInputStream class

Constructor	Does this
`BufferedInputStream(InputStream in)`	It constructs a `BufferedInputStream` object and saves its argument, the input stream in, for later use.
`BufferedInputStream(InputStream in, int size)`	It constructs a `BufferedInputStream` object with the given buffer size.

Table 18.27: Methods of the BufferedInputStream class

Method	Does this
`int available()`	It gets the number of bytes that can be read from this input.
`void close()`	It closes this input stream.
`void mark(int readAheadLimit)`	It sets the current marked position in the stream.
`boolean markSupported()`	It tests whether this input stream supports the `mark()` and `reset()` methods.
`int read()`	It reads the next byte of data.
`int read(byte[] b, int off, int len)`	It reads bytes from this byte input stream into the given byte array, starting at the given offset.
`void reset()`	It resets the buffer to the marked position.
`long skip(long n)`	It skips `n` bytes of input from this input stream.

Here's an example. In this case, we'll remove comments from a `ByteArrayInputStream` object that are delimited with the `#` character. That means when we encounter a `#` character, we'll just stop displaying bytes until we encounter a closing `#`.

For the purposes of this example, we'll only use comments that don't include spaces. Note that sometimes the `#` character is a valid character in text, so if we encounter a `#` and stop displaying text and then encounter a space

before seeing a closing #, the text was not a comment. Therefore, we have to move back to the first # and redisplay the text up to the space.

We'll write this example using the `mark()` and `reset()` methods; we mark a location, then, if needed, use the `reset()` method to go back to that location. Here's what the code looks like (note that you pass a length, in bytes, to the `mark()` method that indicates the maximum number of bytes you'll read before possibly using `reset()`):

```
import java.io.*;
class BufferedInputStreamDemo {
  public static void main(String args[]) throws IOException {
        byte data[] = "Welcome to Dreamtech Press.#comment_goes_here#".getBytes();
        ByteArrayInputStream in = new ByteArrayInputStream(data);
        BufferedInputStream bufferedinputstream = new
        BufferedInputStream(in);
        int character;
        boolean silentflag = false;
        while ((character = bufferedinputstream.read()) != -1) {
                switch(character) { case '#':
                        if (silentflag) { silentflag = false; }
                        else {
                                silentflag = true;
                                bufferedinputstream.mark(100); }
                        break;
                        case ' ':
                        if (silentflag) {
                                silentflag = false;
                                System.out.print("#");
                                bufferedinputstream.reset(); }
                        else
                        System.out.print((char) character);
                        break;
                        default:
                        if (!silentflag) System.out.print((char) character);
                }
        }
  }
}
```

Here's the output of this code: E:\Java8\chapter18\>java BufferedInputStreamDemo

```
Welcome to Dreamtech Press.
```

The *BufferedOutputStream* Class

The `BufferedOutputStream` class is the output counterpart of the `BufferedInputStream` class (see the previous solution). The inheritance diagram for the `BufferedOutputStream` class is as follows:

```
java.lang.Object
  |____java.io.OutputStream
        |____java.io.FilterOutputStream
              |____java.io.BufferedOutputStream
```

You'll find the fields of the `BufferedOutputStream` class in Table 18.28, its constructors in Table 18.29, and its methods in Table 18.30:

Table 18.28: Fields of the BufferedOutputStream class

Field	Does this
`protected byte[] buf`	The internal buffer where data is stored.
`protected int count`	The number of bytes in the buffer.

Table 18.29: Constructors of the BufferedOutputStream class	
Constructor	**Does this**
`BufferedOutputStream(OutputStream out)`	It constructs a new-buffered output stream.
`BufferedOutputStream(OutputStream out, int size)`	It constructs a new buffered output stream to write data with the given buffer size.

Table 18.30: Methods of the BufferedOutputStream class	
Method	**Does this**
`void flush()`	It flushes this buffered output stream.
`void write(byte[] b, int off, int len)`	It writes `len` bytes from the given byte array, starting at offset `off`, to this buffered output stream.
`void write(int b)`	It writes the given byte to this buffered output stream.

The idea behind `BufferedOutputStream` is that you don't have to write data to disk every time a byte is written, because output is buffered. To force a write operation, you can use the `flush()` method.

The *FileInputStream* Class

"I need to open an image file and work with its individual bytes," the NP says. "How can I do that?" "You can use the `FileInputStream` class to work with the bytes in a file," you say. The NP says, "Tell me more!"

The `FileInputStream` class is specially designed to work with byte-oriented input from files and is derived from `InputStream`. The inheritance diagram for the `FileInputStream` class is as follows:

```
java.lang.Object
  |____java.io.InputStream
        |____java.io.FileInputStream
```

You'll find the constructors of the `FileInputStream` class in Table 18.31 and its methods in Table 18.32:

Table 18.31: Constructors of the FileInputStream class	
Constructor	**Does this**
`FileInputStream(File file)`	It constructs a `FileInputStream` object by opening a connection to an actual file
`FileInputStream(FileDescriptor fdObj)`	It constructs a `FileInputStream` object by using the file descriptor `fdObj`
`FileInputStream(String name)`	It constructs a `FileInputStream` object by opening a connection to an actual file given by the file named by `name`

Table 18.32: Methods of the FileInputStream class	
Method	**Does this**
`int available()`	It gets the number of bytes that can be read from this file input stream
`void close()`	It closes this file input stream
`protected void finalize()`	It makes sure that the `close()` method of this file input stream is called when there are no more references to it
`FileDescriptor getFD()`	It gets the `FileDescriptor` object
`int read()`	It reads a byte of data from this input stream
`int read(byte[] b)`	It reads up to `b.length` bytes of data from this input stream called when there are no more references to it into an array of bytes
`int read(byte[ ] b, int off, int len)`	It reads up to `len` bytes of data from this input stream into an array of bytes
`long skip(long n)`	It skips over and discards `n` bytes

All the constructors can throw a `FileNotFoundException`. To create a file input stream, you use the `FileInputStream` class's constructor. The reading methods you use with this class are as follows:

- `int read()`—It reads a byte of data from this input stream and returns it.
- `int read(byte[] b)`—It reads up to `b.length` bytes of data from this input stream into an array of bytes and returns the number of bytes read.
- `int read(byte[] b, int off, int len)`—It reads up to `len` bytes of data from this input stream into an array of bytes and returns the number of bytes read.

Let's consider an example: We open the source code file of the application, read and display 50 bytes of code, skip 50 bytes with the `skip()` method, and then read and display 50 more bytes (note that we can also determine how many bytes are waiting there to be read using the `available()` method, and we can close the stream at the end of the code):

```
import java.io.*;
class FileInputStreamDemo {
  public static void main(String args[]) throws Exception {
        int size;
        FileInputStream fileinputstream = new
        FileInputStream("FileInputStreamDemo.java");
        System.out.println("Available bytes: " + (size
        =fileinputstream.available()));
        System.out.println("Reading 20 bytes....");
        byte bytearray[] = new byte[20];
        if (fileinputstream.read(bytearray) != 20) {
                System.out.println("Could not get 20 bytes"); }
        System.out.println(new String(bytearray, 0, 20));
        System.out.println("Skipping 20 bytes...");
        fileinputstream.skip(20);
        System.out.println("Reading 20 bytes....");
        if (fileinputstream.read(bytearray) != 20) {
                System.out.println("Could not get 20 bytes"); }
        System.out.println(new String(bytearray, 0, 20));
        fileinputstream.close(); }
}
```

The output from this code is shown in Figure 18.5, showing the input to a file:

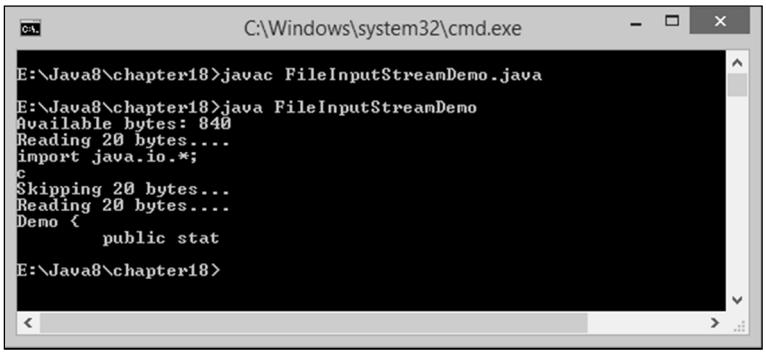


Figure 18.5: The FileInputStreamDemo Class

TIP

An untrusted applet is not allowed to read and write files at all. So, unless you make specific security arrangements, you should not try to read or write files from applets. Therefore, for the sake of convenience, all the programs in this chapter are applications.

The ***FileOutputStream*** Class

"OK," says the Novice Programmer, "I know you use `FileInputStream` to read from files, but what class do you use if you want to write to a file?" You smile and say, "I bet you can guess."

You can use the `FileOutputStream` class to write data, byte by byte, to a file. They can throw a `FileNotFoundException` class or a `SecurityException` class. Remember that the creation of a `FileOutputStream` class is not dependent on the file that already exists. In fact, the `FileOutputStream` class will create the file before opening it for output. An `IOException` is thrown in case you try to open a read-only file. All `FileOutputStream` methods have a void return type.

The inheritance diagram for that class, which is derived from the `OutputStream` class, is as follows:

```
java.lang.Object
  |____java.io.OutputStream
        |____java.io.FileOutputStream
```

The constructors of the `FileOutputStream` class are provided in Table 18.33 and its methods in Table 18.34:

Table 18.33: Constructors of the FileOutputStream class

Constructor	Does this
`FileOutputStream(File file)`	It constructs a file output stream to write to the given `File` object.
`FileOutputStream(File file, boolean append)`	It constructs a file output stream to write to the file represented by the specified `File` object.
`FileOutputStream(FileDescriptor fdObj)`	It constructs an output file stream to write to the given file descriptor.
`FileOutputStream(String name)`	It constructs an output file stream to write to the file with the given name.
`FileOutputStream(String name, boolean append)`	It constructs an output file stream to append to the file with the given name.

Table 18.34: Methods of the FileOutputStream class

Method	Does this
`void close()`	It closes this file output stream.
`protected void finalize()`	It makes sure that the `close()` method of this file output stream is called when there are no more references to this stream.
`FileChannel getChannel()`	It gets the unique `FileChannel` object associated with this file output stream.
`FileDescriptor getFD()`	It gets the file descriptor associated with this stream.
`void write(byte[ ] b)`	It writes `b.length` bytes from the given byte array to this file output stream.
`void write(byte[] b, int off, int len)`	It writes `len` bytes from the given byte array, starting at offset `off`, to this file output stream.
`void write(int b)`	It writes the given byte to this file output stream.

The writing methods you use with this class are as follows:

- `void write(byte[] b)`—Writes `b.length` bytes from the given byte array to this file output stream.
- `void write(byte[] b, int off, int len)`—Writes `len` bytes from the given byte array, starting at offset, to the file output stream.
- `void write(int b)`—Writes the given byte to the file output stream.

Here's an example in which we write the string of text "This is a string of text." to a file in three ways—byte by byte, all at once, and in part. Here's what the code looks like:

```
import java.io.*;
class FileOutputStreamDemo {
  public static void main(String args[]) throws Exception {
        byte data[] = "This is a string of text.".getBytes();
        FileOutputStream fileoutputstream1 = new
        FileOutputStream("file1.txt");
        for (int loop_index = 0; loop_index < data.length; loop_index++) {
```

```
                    fileoutputstream1.write(data[loop_index]); }
            FileOutputStream fileoutputstream2 = new
            FileOutputStream("file2.txt");
            fileoutputstream2.write(data);
            FileOutputStream fileoutputstream3 = new
            FileOutputStream("file3.txt");
            fileoutputstream3.write(data, 5, 10);
            fileoutputstream1.close();
            fileoutputstream2.close();
            fileoutputstream3.close(); }
    }
```

Compile and execute this code and find three files-file1.txt, file2.txt, and file3.txt that are created.

Working with the **Reader** Class

"OK," says the Novice Programmer, "I'm going to work with characters, instead of just bytes including Unicode. How do I handle that?" "With the `Reader` and `Writer` classes," you say. "Tell me more!" says the NP.

You use the `Reader` and `Writer` classes to read and write character-streams. As with `InputStream` and `OutputStream`, you base classes on `Reader` and `Writer` instead of using them directly. The `Reader` implements the `Closeable` and `Readable` interfaces. All the methods of this class throw an `IOException` on error conditions. Here's the inheritance diagram for `Reader`:

```
java.lang.Object
  |____java.io.Reader
```

You'll find the field of the `Reader` class in Table 18.35, its constructors in Table 18.36, and its methods in Table 18.37:

Table 18.35: The field of the Reader class

Field	Does this
`protected Object lock`	The lock used to synchronize operations.

Table 18.36: Constructors of the Reader class

Constructor	Does this
`protected Reader()`	It constructs a new character-stream.
`protected Reader(Object lock)`	It constructs a new character stream reader whose critical sections will synchronize on the given lock.

Table 18.37: Methods of the Reader class

Method	Does this
`abstract void close()`	It closes the stream.
`void mark(int readAheadLimit)`	It marks the present position in the stream.
`boolean markSupported()`	It tells whether this stream supports the `mark()` operation.
`int read()`	It reads a single character.
`int read(char[ ] cbuf)`	It reads characters into an array.
`abstract int read(char[] cbuf, int off, int len)`	It reads characters into a portion of an array.
`boolean ready()`	It tells whether this stream is ready to be read.
`void reset()`	It resets the stream.
`long skip(long n)`	It skips characters.

Working with the **Writer** Class

The `Writer` class is the character-based counterpart of the `Reader` class (see the previous solution). Here's the inheritance diagram for `Writer`:

```
java.lang.Object
  |____java.io.Writer
```

The `Writer` implements the `Closeable`, `Flushable`, and `Appendable` interfaces. A void value is returned and an `IOException is thrown` by all the methods in this class in the event of errors. You'll find the field of the `Writer` class in Table 18.38, its constructors in Table 18.39, and its methods in Table 18.40:

Table 18.38: The field of the Writer class

Field	Does this
`protected Object lock`	The lock used to synchronize operations on this stream.

Table 18.39: Constructors of the Writer class

Constructor	Does this
`protected Writer()`	It creates a new character-stream writer.
`protected Writer(Object lock)`	It creates a new character-stream writer whose critical sections will synchronize on the given lock.

Table 18.40: Methods of the Writer class

Method	Does this
`Writer append(char char)`	It appends the character to the end of the output stream.
`Writer append(CharSequence chars)`	It appends the string to the end of the output stream.
`Writer append(CharSequence csq, int start, int end)`	It appends a subsequence of the specified character sequence to this writer.
`abstract void close()`	It closes the stream, flushing it first.
`abstract void flush()`	It flushes the stream.
`void write(char[] cbuf)`	It writes an array of characters.
`abstract void write(char[ ] cbuf, int off, int len)`	It writes a portion of an array of characters.
`void write(int c)`	It writes a single character.
`void write(String str)`	It writes a string.
`void write(String str, int off, int len)`	It writes a portion of a string.

Accepting Input from the Keyboard with the **InputStreamReader** Class

"Well," says the Novice Programmer, "all I want to do is read input from the keyboard. Can I use `System.in`?" "Yes," you say, "but construct an `InputStreamReader` stream from it first." "Oh," the NP says.

You usually construct an `InputStreamReader` stream from `System.in` for reading keys from the keyboard. After that, you can use the `read()` method of the `InputStreamReader` class for reading the text typed by the user. Here's the inheritance diagram for `InputStreamReader`:

```
java.lang.Object
  |____java.io.Reader
        |____java.io.InputStreamReader
```

You'll find the constructors of the `InputStreamReader` class in Table 18.41 and its methods in Table 18.42:

Table 18.41: Constructors of the InputStreamReader class

Constructor	Does this
`InputStreamReader(InputStream in)`	It creates an `InputStreamReader` object.
`InputSreamReader(InputStream in, Charset cs)`	It creates an `InputStreamReader` that uses the given charset.
`InputSreamReader(InputStream in, CharsetDecoder dec)`	It creates an `InputStreamReader` that uses the given charset decoder.
`InputStreamReader(InputStream in, String enc)`	It creates an `InputStreamReader` object that uses the named character encoding.

Table 18.42: Methods of the InputStreamReader class

Method	Does this
`void close()`	It closes the stream.
`String getEncoding()`	It gets the name of the character encoding.
`int read()`	It reads a single character.
`int read(char[ ] cbuf, int off, int len)`	It reads characters into a portion of an array.
`boolean ready()`	It returns True if this stream is ready to be read.

Here's an example: We'll just read keys from the keyboard and echo them. Note that the input stream does not become available until the user presses the Enter key. Here's what the code looks like:

```
import java.io.*;
class InputStreamReaderDemo
{
  public static void main(String args[])
  {
        try {
               String character;
               InputStreamReader inputstreamreader = new InputStreamReader(System.in);
               BufferedReader br = new BufferedReader(inputstreamreader);
               character = br.readLine();
               System.out.print(character);
            }
            catch (IOException e) { }
  }

}
```

The output of this code is shown in Figure 18.6, printing the string taken from the keyboard as an input:

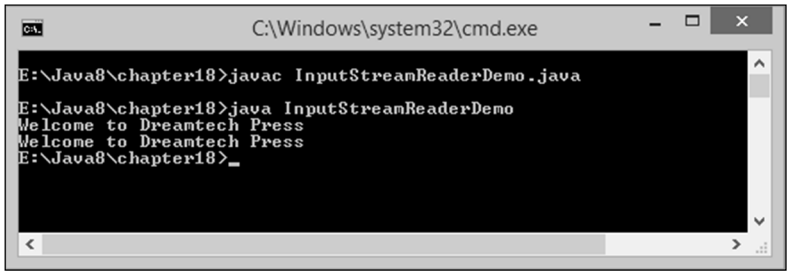


Figure 18.6: Taking input from the user

Note that the echoed line appears after the user enters text and presses the Enter key.

Working with the **OutputStreamWriter** Class

The `OutputStreamWriter` class works as a bridge from character streams to byte streams. This class performs the encoding of characters into bytes with the help of a specified character set. Here's the inheritance diagram for `OutputStreamWriter`:

```
java.lang.Object
  |____java.io.Writer
        |____java.io.OutputStreamWriter
```

You'll find the constructors and methods of the `OutputStreamWriter` class in Table 18.43 and Table 18.44, respectively:

Table 18.43: Constructors of the OutputStreamWriter class

Constructor	Does this
`OutputStreamWriter(OutputStream out)`	It constructs an `OutputStreamWriter` object.
`OutputSreamWriter(OutputStream out, Charset cs)`	It creates an `OutputStreamWriter` that uses the given charset.
`OutputSreamWriter(OutputStream out, CharsetEncoder enc)`	It creates an `OutputStreamWriter` that uses the given charset encoder.
`OutputStreamWriter(OutputStream out, String enc)`	It constructs an `OutputStreamWriter` object that uses the named character encoding.

Table 18.44: Methods of the OutputStreamWriter class

Method	Does this
`void close()`	It closes the stream.
`void flush()`	It flushes the stream.
`String getEncoding()`	It gets the name of the character encoding being used by this stream.
`void write(char[ ] cbuf, int off, int len)`	It writes a section of an array of characters.
`void write(int c)`	It writes a single character.
`void write(String str, int off, int len)`	It writes a portion of a string.

Working with Files

After learning about the streams which provide a simple model for reading and writing data that supported all the operations common with a disk file, you will learn how to work with a file using the non-stream file I/O.

The `File` class handles files that are machine-dependent in an independent manner, that is, you can create platform-independent code easily that keeps monitoring and manipulating files using the `File` class. This class is available in the `java.io` package.

*Using the **File** Class*

"Hmm," says the Novice Programmer, "little bit of an embarrassment—my program tried to open a file that didn't exist, right when I was running it for the Big Boss." "Well," you say, "why didn't you first test whether the file existed using the `File` class's `exists()` method?" The NP says, "Now you tell me!"

Except for the `File` class, most of the classes that are defined by `java.io` operate on streams, i.e., they deal directly with files and the file system. The `File` class does not state how information is accessed from or stored in files. Instead, it describes the properties of a file itself.

You can use the `File` class to determine a lot about files, such as their sizes; whether you can write to them; whether an item is a file, directory, or named pipe; and a lot more for manipulating the information associated with a disk file, such as permissions, time, date, and for navigating subdirectory hierarchies. Here's the inheritance diagram for the `File` class:

```
java.lang.Object
  |____java.io.File
```

A Java directory is considered to be an additional property, in which an array of filenames can be verified by using the `list()` method. You'll find the fields for the `File` class in Table 18.45, its constructors in Table 18.46, and its methods in Table 18.47:

Table 18.45: Fields of the File class

Field	Does this
`static String pathSeparator`	It is a system-dependent path-separator string.
`static char pathSeparatorChar`	It is a system-dependent path-separator character.
`static String separator`	It is a system-dependent default name-separator character, represented as a string for convenience.
`static char separatorChar`	It is a system-dependent default name-separator character.

Table 18.46: Constructors of the File class

Constructor	Does this
`File(File parent, String child)`	It constructs a new `File` instance from a parent pathname and a child pathname string.
`File(String pathname)`	It constructs a new `File` instance by converting the given pathname string into an abstract pathname.
`File(String parent, String child)`	It constructs a new `File` instance from a parent pathname string and a child pathname string.
`File(URI uri)`	It constructs a new `File` instance by converting the given file:URI into an abstract pathname.

Table 18.47: Methods of the File class

Method	Does this
`boolean canExecute()`	It returns True if the application can execute the file denoted by this abstract pathname.
`boolean canRead()`	It returns True if the application can read the file given by this abstract pathname.
`boolean canWrite()`	It returns True if the application can modify the file given by this abstract pathname.
`int compareTo(File pathname)`	It compares two abstract pathnames.
`int compareTo(Object o)`	It compares this abstract pathname to another object.
`boolean createNewFile()`	It constructs a new, empty file.
`static File createTempFile(String prefix, String suffix)`	It constructs an empty file in the default temporary file directory.
`static File createTempFile(String prefix, String suffix, File directory)`	It constructs a new, empty file in the given directory.
`boolean delete()`	It deletes the file or directory.
`void deleteOnExit()`	It requests that the file or directory given by this pathname be deleted when the virtual machine exits.

Table 18.47: Methods of the File class

Method	Does this
`boolean equals(Object obj)`	It compares this pathname for equality with the given object.
`boolean exists()`	It compares whether the file given by this abstract pathname exists.
`File getAbsoluteFile()`	It gets the absolute form of this abstract pathname.
`String getAbsolutePath()`	It gets the absolute pathname string of this abstract pathname.
`File getCanonicalFile()`	It gets the canonical form of this abstract pathname.
`String getCanonicalPath()`	It gets the canonical pathname string of this abstract pathname.
`long getFreeSpace()`	It gets the number of unallocated bytes in the partition named by this abstract pathname.
`String getName()`	It gets the name of the file or directory given by this abstract pathname.
`String getParent()`	It gets the pathname string of this abstract pathname's parent or gets null if this pathname does not name a parent directory.
`File getParentFile()`	It gets the abstract pathname of this abstract pathname's parent or gets null if this pathname does not name a parent directory.
`String getPath()`	It converts this abstract pathname into a pathname string.
`long getTotalSpace()`	It gets the size of the partition named.
`long getUsableSpace()`	It gets the number of bytes available on the partition named.
`int hashCode()`	It computes a hash code for this abstract pathname.
`boolean isAbsolute()`	It returns True if this abstract pathname is absolute.
`boolean isDirectory()`	It returns True if the file given by this abstract pathname is a directory.
`boolean isFile()`	It returns True if the file given by this abstract pathname is a normal file.
`boolean isHidden()`	It returns True if the file named by this abstract pathname is a hidden file.
`long lastModified()`	It returns the time that the file given by this abstract pathname was last modified.
`long length()`	It returns the length of the file given by this abstract pathname.
`String[ ] list()`	It returns an array of strings naming the files and directories in the directory given by this abstract pathname.
`String[ ] list(FilenameFilter filter)`	It returns an array of strings naming the files and directories in the directory given by this pathname that satisfies the given filter.
`File[ ] listFiles()`	It returns an array of abstract pathnames indicating the files in the directory.
`File[ ] listFiles(FileFilter filter)`	It returns an array of abstract pathnames indicating the files and directories in the directory that fulfills the given filter.
`File[ ] listFiles(FilenameFilter filter)`	It accepts the `FilenameFilter` object and returns an array of abstract pathnames indicating the files and directories in the directory that fulfills the given filter.
`static File[ ] listRoots()`	It lists the available file system roots.
`boolean mkdir()`	It creates the directory named by this abstract pathname.
`boolean mkdirs()`	It creates the directory named by this pathname, including any necessary but nonexistent parent directories.
`boolean renameTo(File dest)`	It renames the file given by this pathname.
`boolean setExecutable(boolean executable)`	It sets owner's execute permission for this abstract pathname.

Table 18.47: Methods of the File class

Method	Does this
`boolean setExecutable (boolean executable, boolean ownerOnly)`	It sets owner's or everybody's execute permission for this abstract pathname.
`boolean setLastModified(long time)`	It sets the last modified time of the file or directory.
`boolean setReadable(boolean readable)`	It sets owner's read permission for this abstract pathname.
`boolean setReadable(boolean readable, boolean ownerOnly)`	It sets read permission for the owner of the file or other users for the given abstract pathname.
`boolean setReadOnly()`	It marks the file or directory named by this pathname so that it can only be read.
`boolean setWritable(boolean writable)`	It sets owner's write permission for this abstract pathname.
`boolean setWritable(boolean writable, boolean ownerOnly)`	It sets owner's or everybody's write permission for this abstract pathname.
`String toString()`	It gets the pathname string of this abstract pathname.
`URI toURI()`	It constructs a file URI that represents this abstract pathname.
`URL toURL()`	It converts this abstract pathname into a URL. It is recommended that abstract pathname should be converted to URI first using the `toURI()` method and then use `URI.toURL()` as it helps in escaping illegal characters.

Note that you can use `isFile()` to determine whether an item is a true file or something else, such as a directory (which you can check with `isDirectory()`) or a named pipe. In other words, the `isFile()` method returns true if called on a file (and behaves like one) and false if called on a directory. Use `toURL()` to convert a file's path into a URL, rename files, and do more with the `File` class. The `isAbsolute()` method returns true if the file has an absolute path and false if it is relative.

Let's look at an example. Here, we put several of the methods of the `File` class to work, getting some information about a file named `file.txt`. Here's what the code looks like:

```
import java.io.File;
class FileDemo {
  public static void main(String args[]) {
        File file1 = new File("file.txt");
        System.out.println("File: " + file1.getName() + (file1.isFile() ?
        " is a file" : " is a named pipe or directory"));
        System.out.println("Size: " + file1.length());
        System.out.println("Path: " + file1.getPath());
        System.out.println("Parent: " + file1.getParent());
        System.out.println("Absolute Path: " + file1.getAbsolutePath());
        System.out.println("File was last modified: " +
        file1.lastModified());
        System.out.println(file1.exists() ? "File exists" :
        "File does not exist");
        System.out.println(file1.canRead() ? "File can be read from" :
        "File cannot be read from");
        System.out.println(file1.canWrite() ? "File can be written to" :
        "File cannot be written to");
        System.out.println(file1.isDirectory() ? "File is a directory" :
        "File is not a directory");
        System.out.println(file1.isFile() ? "is a true file" :
        "is a special file");
```

```
            System.out.println(file1.isAbsolute() ? "is absolute" :
            "is not absolute"); }
    }
```

Create a file named `file.txt` before executing this code file in the same location where you are saving Java and class files. The result of this code is shown in Figure 18.7 (note that the time the file was last modified is returned in the usual fashion—in milliseconds since 1/1/70—which you can convert into a more readable date with the `Date` class):

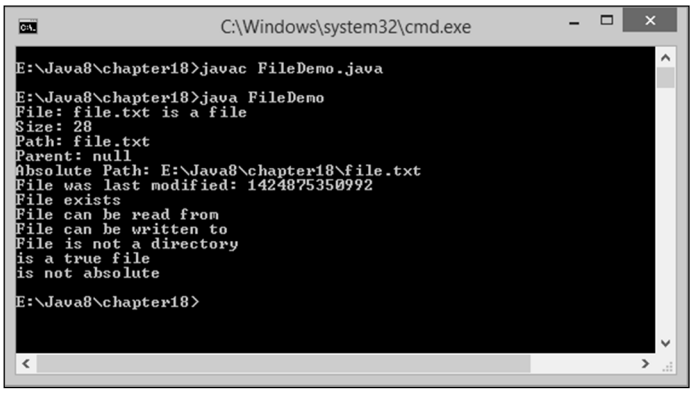


Figure 18.7: Creating a file

In the preceding case, `getName()` returns the name of the file, `getParent()` returns the name of the parent directory, and `exists()` returns true.

The `File` class is not symmetrical, which signifies that there are many methods that allow you to monitor the properties of a simple file, though no simultaneous function exists to change those attributes. This class also includes two useful utility methods:

- `boolean renameTo(File newname)`—Here `newname` refers to the new filename. This will return true upon success and false if the directory cannot be renamed.
- `boolean delete()`—This utility method deletes a directory if the directory is empty. This will return true if it deletes the file and false if the file cannot be deleted.

As you can see, using the `File` class, you can determine a great deal about files and directories.

Using the ***FileReader*** Class

"How can I read a character-based file?" the NP asks. "Well," you say, "probably the best way is with the `FileReader` class." "Great," says the NP, "let's write some code!"

You use the `FileReader` class to create a character-based stream that reads from a file. Here's the inheritance diagram for this class:

```
java.lang.Object
   |____java.io.Reader
         |____java.io.InputStreamReader
                  |____java.io.FileReader
```

This class only defines constructors in addition to the functionality it inherits from `InputStreamReader`, and you'll find them in Table 18.41. All the constructors can throw a `FileNotFoundException`.

Here's an example. In this case, we'll open a file, `file.txt`, having the contents "Welcome to Dreamtech Press", read that file's data into a buffer, and then display the data:

```
import java.io.*;
class FileReaderDemo
{
   public static void main(String args[]) throws Exception
   {
         FileReader filereader = new FileReader("file.txt");
         char data[] = new char[1024];
```

```
            int charsread = filereader.read(data);
            System.out.println(new String(data, 0, charsread));
            filereader.close();
        }
    }
```

The output of this program is shown in Figure 18.8, reading the file:

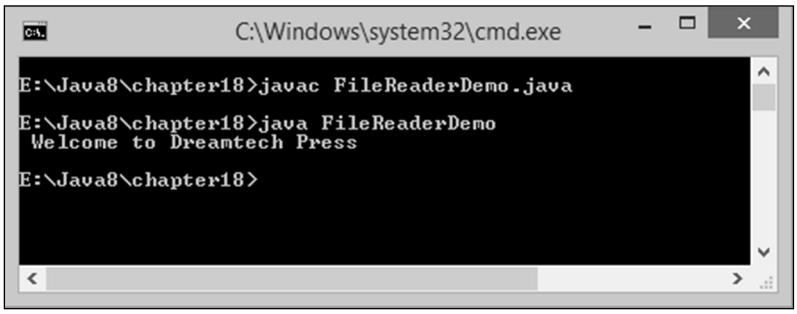


Figure 18.8: Reading the text from the file

As you can see, working with `FileReader` is relatively a simple process in Table 18.48:

Table 18.48: Constructors of the FileReader class

Constructor	Does this
`FileReader(File file)`	It constructs a `FileReader` object
`FileReader(FileDescriptor fd)`	It constructs a `FileReader` object from a file descriptor
`FileReader(String fileName)`	It constructs a `FileReader` object from a file name

*Using the **FileWriter** Class*

The `FileWriter` class is the character-based writer counterpart of the `FileReader` class (see the previous solution for more details). Here's the inheritance diagram for the `FileWriter` class:

```
java.lang.Object
  |____java.io.Writer
        |____java.io.OutputStreamWriter
              |____java.io.FileWriter
```

This class only defines constructors in addition to the functionality it inherits from `OutputStreamWriter`, and you'll find them in Table 18.49:

Table 18.49: Constructors of the FileWriter class

Constructor	Does this
`FileWriter(File file)`	It constructs a `FileWriter` object from a `File` object
`FileWriter(File file, boolean append)`	It constructs a `FileWriter` object given a `File` object
`FileWriter(FileDescriptor fd)`	It constructs a `FileWriter` object from a file descriptor
`FileWriter(String fileName)`	It constructs a `FileWriter` object from a file name
`FileWriter(String fileName, boolean append)`	It constructs a `FileWriter` object for appending

The constructors can throw an `IOException`.

Creation of a `FileWriter` is not dependent on the file that already exists. It creates the file before opening it for output when an object is already created. An `IOException` is thrown in case you attempt to open a read-only file. Here's an example: We'll write data from a character buffer to a file, character by character, all at once, and in part, using `FileWriter`:

```
import java.io.*;
class FileWriterDemo {
  public static void main(String args[]) throws Exception {
        char data[] = {'T','h','i','s',' ','i','s',' ','a',
        ' ','s','t','r','i','n','g',' ','o','f',
        ' ','t','e','x','t','.' };
        FileWriter filewriter1 = new FileWriter("f1.txt");
        for (int loop_index = 0; loop_index < data.length; loop_index++) {
              filewriter1.write(data[loop_index]); }
        FileWriter filewriter2 = new FileWriter("f2.txt");
        filewriter2.write(data);
        FileWriter filewriter3 = new FileWriter("f3.txt");
        filewriter3.write(data, 5, 10);
        filewriter3.append("Dreamtech Press");
        filewriter1.close();
        filewriter2.close();
        filewriter3.close(); }
}
```

After executing the code, three files named f1.txt, f2.txt and f3.txt will be created in the same location where you have saved the .java file.

Working with the **RandomAccessFile** Class

Using the `RandomAccessFile` class, you can move around in a file using the `seek()` method. That's what random access is all about—being able to move around in a file at random. The inheritance diagram for the `RandomAccessFile` class is as follows:

```
java.lang.Object
  |____java.io.RandomAccessFile
```

`RandomAccessFile` is not the child class of `InputStream` or `OutputStream`; instead, it encapsulates a random-access file. It also implements the `Closeable` interface. `RandomAccessFile` supports positioning requests, i.e., you can position the file pointer within the file.

With random access comes the type of access:

- `r`—File is read and not written.
- `rw`—File is opened in the read-write mode.
- `rws`—File is opened for read-write operation and every change to the content/metadata needs to be written synchronously to the underlying storage device.
- `rwd`—File is opened for read-write operation and every change to the content needs to be written synchronously to the underlying storage device.

The `seek()` method is used to set the current position of the file pointer within the file. You'll find the constructors of the `RandomAccessFile` class in Table 18.50 and its methods in Table 18.51:

Table 18.50: Constructors of the RandomAccessFile class

Constructor	Does this
`RandomAccessFile(File file, String mode)`	It constructs a random-access file stream to read from and optionally to write to the file given by the `File` argument.
`RandomAccessFile(String name, String mode)`	It constructs a random-access file stream to read from and optionally to write to a file with the given name.

Table 18.51: Methods of the RandomAccessFile class

Method	Does this
`void close()`	It closes this random-access file stream.
`FileChannel getChannel()`	It gets the unique File Channel object associated with this file.
`FileDescriptor getFD()`	It gets the file descriptor object associated with this stream.

Table 18.51: Methods of the RandomAccessFile class	
Method	**Does this**
`long getFilePointer()`	It gets the current offset in this file.
`long length()`	It gets the length of this file.
`int read()`	It gets a byte of data from this file.
`int read(byte[] b)`	It gets up to `b.length` bytes of data from this file into an array of bytes.
`int read(byte[ ] b, int off, int len)`	It gets up to `len` bytes of data from this file into an array of bytes.
`boolean readBoolean()`	It gets a `boolean` from this file.
`byte readByte()`	It gets a signed 8-bit value from this file.
`char readChar()`	It gets a Unicode character from this file.
`double readDouble()`	It gets a `double` from this file.
`float readFloat()`	It gets a `float` from this file.
`void readFully(byte[ ] b)`	It gets `b.length` bytes from this file into the byte array, starting at the current file pointer.
`void readFully(byte[] b, int off, int len)`	It gets exactly `len` bytes from this file into the byte array, starting at the current file pointer.
`int readInt()`	It gets a signed 32-bit integer from this file.
`String readLine()`	It gets the next line of text from this file.
`long readLong()`	It gets a signed 64-bit integer from this file.
`short readShort()`	It gets a signed 16-bit number from this file.
`int readUnsignedByte()`	It gets an unsigned 8-bit number from this file.
`int readUnsignedShort()`	It gets an unsigned 16-bit number from this file.
`String readUTF()`	It gets a string from this file.
`void seek(long pos)`	It sets the file-pointer offset at which the next read or write happens.
`void setLength(long newLength)`	It sets the length of this file.
`int skipBytes(int n)`	It attempts to skip over `n` bytes of input.
`void write(byte[] b)`	It writes `b.length` bytes from the given byte array to this file, starting at the file pointer.
`void write(byte[ ] b, int off, int len)`	It writes `len` bytes from the given byte array, starting at offset `off`, to this file.
`void write(int b)`	It writes the given byte to this file.
`void writeBoolean(boolean v)`	It writes a `boolean` to the file as a 1-byte value.
`void writeByte(int v)`	It writes a `byte` to the file as a 1-byte value.
`void writeBytes(String s)`	It writes the `string` to the file as a sequence of bytes.
`void writeChar(int v)`	It writes a `char` to the file as a 2-byte value, high byte first.
`void writeChars(String s)`	It writes a `String` to the file as a sequence of characters.
`void writeDouble(double v)`	It converts the `double` argument to a long and then writes that long value to the file.
`void writeFloat(float v)`	It converts the `float` argument to an `int` and then writes that `int` value to the file.
`void writeInt(int v)`	It writes an `int` to the file as 4 bytes, high byte first.

Table 18.51: Methods of the RandomAccessFile class	
Method	**Does this**
`void writeLong(long v)`	It writes a `long` to the file as 8 bytes, high byte first.
`void writeShort(int v)`	It writes a `short` to the file as 2 bytes, high byte first.
`void writeUTF(String str)`	It writes a `string` to the file using UTF-8 encoding.

Note that one of the attractions of the `RandomAccessFile` class is that you can both read and write using this class, without having to break things up into input and output objects.

Working with Character Arrays

A string is a contiguous array of characters, so different operations can be performed on a string by using the array of characters. The operations performed on strings include dividing them up into parts called substrings, combining them together to create new strings, and replacing parts of them with other strings.

*Using the **CharArrayReader** Class*

"You can use `ByteArrayInputStream` and `ByteArrayOutputStream` to read and write bytes from an array," the NP says, "but what about characters? Are there `CharArrayInputStream` and `CharArrayOutputStream` classes?" "No," you say. "Remember that you use readers and writers with character-streams, which means you should use `CharArrayReader` and `CharArrayWriter` instead."

You use `CharArrayReader` to read from a character array. Here's the inheritance diagram for this class:

```
java.lang.Object
  |____java.io.Reader
        |____java.io.CharArrayReader
```

You'll find the fields of the `CharArrayReader` class in Table 18.52, its constructors in Table 18.53, and its methods in Table 18.54:

Table 18.52: Fields of the CharArrayReader class	
Field	**Does this**
`protected char[] buf`	The character buffer.
`protected int count`	The number of characters in the buffer.
`protected int markedPos`	The position of the current mark in the buffer.
`protected int pos`	The current buffer position.

Table 18.53: Constructors of the CharArrayReader class	
Constructor	**Does this**
`CharArrayReader(char[] buf)`	It constructs a `CharArrayReader` object from the given array of characters.
`CharArrayReader(char[] buf, int offset, int length)`	It constructs a `CharArrayReader` object from the given array of characters.

Table 18.54: Methods of the CharArrayReader class	
Method	**Does this**
`void close()`	It closes the stream.
`void mark(int readAheadLimit)`	It marks the present position in the stream.
`boolean markSupported()`	It returns True if this stream supports the `mark` operation.
`int read()`	It reads a single character.

Table 18.54: Methods of the CharArrayReader class	
Method	**Does this**
`int read(char[ ] b, int off, int len)`	It reads characters into a portion of an array.
`boolean ready()`	It tells whether this stream is ready to be read
`void reset()`	It resets the stream to the most recent mark
`long skip(long n)`	It skips characters

Here's an example. In this case, we just read characters from an array and print them out:

```
import java.io.*;
public class CharArrayReaderDemo {
   public static void main(String args[]) throws IOException {
         char data[] = {'T','h','i','s',' ','i','s',' ','a',
                 ' ','s','t','r','i','n','g',' ','o','f',
                 ' ','t','e','x','t','.' };
         CharArrayReader chararrayreader = new CharArrayReader(data);
         int character;
         while((character = chararrayreader.read()) != -1) {
                 System.out.print((char)character); }
   }
}
```

Here's the output of this code:

```
E:\Java8\chapter18\>java CharArrayReaderDemo
This is a string of text.
```

Take the case of this example which constructs one object using the entire sentence, while the other object has only the first six letters:

```
import java.io.*;
public class CharArrayReaderDemo1 {
   public static void main(String args[]) throws IOException {
         String t = "The quick brown fox ran over the mount";
         int l = t.length();
         char w[] = new char[l];
         t.getChars(0, l, w, 0);
         CharArrayReader input1 = new CharArrayReader(w);
         CharArrayReader input2 = new CharArrayReader(w, 0, 6);
         int j;
         System.out.println("All the characters:");
         while((j = input1.read()) != -1) { System.out.print((char)j); }
         System.out.println();
         System.out.println("First 6 characters:");
         while((j = input2.read()) != -1) { System.out.print((char)j); }
         System.out.println(); }
}
```

The result of the code is shown in Figure 18.9:

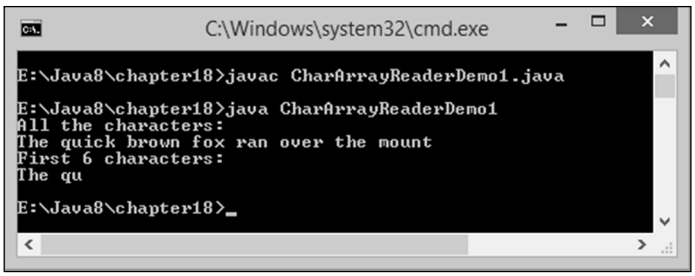


Figure 18.9: Writing the characters from the array

Using the **CharArrayWriter** Class

The `CharArrayWriter` class is the writer counterpart of the `CharArrayReader` class (see the previous solution). You use this class to write to character arrays. Here's the inheritance diagram for this class:

```
java.lang.Object
  |____java.io.Writer
        |____java.io.CharArrayWriter
```

You'll find the fields of the `CharArrayWriter` class in Table 18.55, its constructors in Table 18.56, and its methods in Table 18.57:

Table 18.55: Fields of the CharArrayWriter class

Field	Does this
`protected char[] buf`	The buffer where data is stored.
`protected int count`	The number of characters in the buffer.

Table 18.56: Constructors of the CharArrayWriter class

Constructor	Does this
`CharArrayWriter()`	It constructs a new `CharArrayWriter` object.
`CharArrayWriter(int initialSize)`	It constructs a new `CharArrayWriter` object with the given initial size.

Table 18.57: Methods of the CharArrayWriter class

Method	Does this
`CharArrayWriter append(char c)`	It appends the specified character to this writer.
`CharArrayWriter append (CharSequence csq)`	It appends the specified character sequence to this writer.
`CharArrayWriter append (CharSequence csq, int start, int end)`	It appends a subsequence of the specified character sequence to this writer.
`void close()`	It closes the stream.
`void flush()`	It flushes the stream.
`void reset()`	It resets the buffer.
`int size()`	It gets the current size of the buffer.
`char[] toCharArray()`	It gets a copy of the input data.
`String toString()`	It converts input data to a string.
`void write(char[ ] c, int off, int len)`	It writes characters to the buffer.
`void write(int c)`	It writes a character to the buffer.
`void write(String str, int off, int len)`	It writes a portion of a string to the buffer.
`void writeTo(Writer out)`	Writes contents of buffer to another character-stream.

Let's concentrate on two constructors. In the first form, a buffer with a default size is created. In the second, a buffer is created with a size equal to that specified by `initialSize`. The `buf` field of the `CharArrayWriter` class stores the buffer, which will be increased automatically. The number of characters stored in the buffer is contained in the `count` field of the `CharArrayWriter` class. Both `buf` and `count` are protected fields.

Look at this example which demonstrates this concept of `CharArrayWriter`:

```
import java.io.*;
class CharArrayWriterDemo {
  public static void main(String args[]) throws IOException {
        CharArrayWriter f = new CharArrayWriter();
        String W = "I/O stream classes have become a thicket in Java";
        char buf[] = new char[W.length()];
        W.getChars(0, W.length(), buf, 0);
        f.write(buf);
        System.out.println("\nBuffer of default size:");
        System.out.println(f.toString());
        System.out.println("\n\nWhat goes into the array-->");
        char A[] = f.toCharArray();
        for (int j=0; j<A.length; j++) { System.out.print(A[j]); }
        System.out.println("\n\nFinally going to FileWriter()-->");
        FileWriter f2 = new FileWriter("cofbean.txt");
        f.writeTo(f2); f2.close();
        System.out.println("\n........ after a quick reset");
        f.reset();
        for (int j=0; j<3; j++)
                f.write('0'); System.out.println(f.toString());
  }
}
```

This code will create a file `cofbean.txt` in the current directory to store the string 'I/O stream classes have become a thicket in Java'. But after the call to `reset()`, the three 0's end up in the beginning. See the output of the code in Figure 18.10:

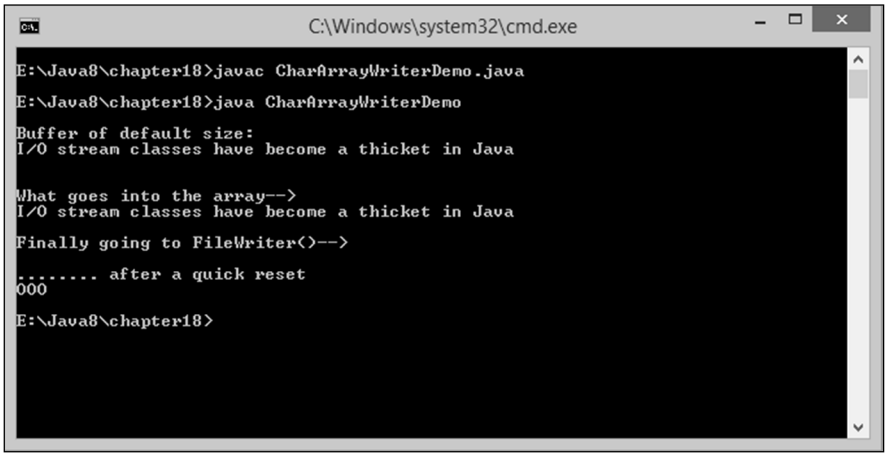


Figure 18.10: Using the CharacterArrayWriter class

Working with Buffers

In the `java.io` package, a file or stream is a sequence of bytes or byte array(s). The `DataInput` and `DataOutput` interfaces can help out, but they work on streams and don't allow byte order to be specified. Here, the NIO Buffer class and its subclasses come in use. A commonly used buffer class is `ByteBuffer`.

It allows a byte array (or block of memory/file) to be managed as a buffer with methods to deal with the current read/write position, maximum position, and array of bytes.

*Using the **BufferedReader** Class*

"My files are line oriented," says the Novice Programmer. "Is there some easy way to read them in line by line instead of character by character?" "Certainly," you say, "you can use the `readLine()` method of the `BufferedReader` class." "Great." says the NP. "You write the code, I'll watch."

The `BufferedReader` class provides a buffered character reader class and the `readLine()` method. Here's the inheritance diagram for `BufferedReader`:

```
java.lang.Object
  |____java.io.Reader
        |____java.io.BufferedReader
```

The `BufferedReader` class consists of two constructors. In the first one, a buffered character-stream uses a default buffer size. In the second, the size of the buffer is passed in `sz`**.** You'll find the constructors of the `BufferedReader` class in Table 18.58 and its methods in Table 18.59:

Table 18.58: Constructors of the BufferedReader class

Constructor	Does this
`BufferedReader(Reader in)`	It constructs a buffering character input stream.
`BufferedReader(Reader in, int sz)`	It constructs a buffering character input stream that uses an input buffer of the given size.

Table 18.59: Methods of the BufferedReader class

Method	Does this
`void close()`	It closes the stream.
`void mark(int readAheadLimit)`	It marks the present position in the stream.
`boolean markSupported()`	It returns True if this stream supports the `mark` operation.
`int read()`	It reads a single character.
`int read(char[ ] cbuf, int off, int len)`	It reads characters of a portion of an array.
`String readLine()`	It reads a line of text.
`boolean ready()`	It tells whether this stream is ready to be read.
`void reset()`	It resets the stream to the most recent mark.
`long skip (long n)`	It skips characters.

Here's another example using the `BufferedReader` class. In this case, we read the file `file4.txt` line by line with the `readLine()` method. Here's the content of that file:

```
This is a wonderful program where you can find the capability of File() class.
Some streams provide data buffering.
And some do not.
Some are designed to work with data in memory buffers.
Some with files.
```

Here's the code:

```
import java.io.*;
class BufferedReaderDemo
{
  public static void main(String args[]) throws Exception
  {
        FileReader filereader = new FileReader("file4.txt");
        BufferedReader bufferedreader = new BufferedReader(filereader);
        String instring;
        while((instring = bufferedreader.readLine()) != null) {
                System.out.println(instring); } filereader.close(); }
}
```

Here's the output of the code when it's run:

```
E:\Java8\chapter18\>java BufferedReaderDemo
This is a wonderful program where you can find the capability of File() class.
Some streams provide data buffering.
And some do not.
Some are designed to work with data in memory buffers.
Some with files.
```

Using the **BufferedWriter** Class

The `BufferedWriter` class is the writer counterpart of the `BufferedReader` class (see the previous solution). Here's the inheritance diagram for the `BufferedWriter` class:

```
java.lang.Object
  |____java.io.Writer
        |____java.io.BufferedWriter
```

You'll find the constructors of the `BufferedWriter` class in Table 18.60 and its methods in Table 18.61:

Table 18.60: Constructors of the BufferedWriter class

Constructor	Does this
`BufferedWriter(Writer out)`	It constructs a buffered character-output stream.
`BufferedWriter(Writer out, int sz)`	It constructs a new-buffered character-output stream that uses an output buffer of the given size.

Table 18.61: Methods of the BufferedWriter class

Method	Does this
`void close()`	It closes the stream.
`void flush()`	It flushes the stream.
`void newLine()`	It writes a line separator.
`void write(char[] cbuf, int off, int len)`	It writes a section of an array of characters.
`void write(int c)`	It writes a single character.
`void write(String s, int off, int len)`	It writes a portion of a string.

In Table 18.60, the first form of constructor creates a buffered stream with a default size and in the second constructor, the size of the buffer is passed in `sz`.

Working with the **PushbackReader** Class

The `PushbackReader` class enables characters to be pushed back into the stream. When you use this class's `unread()` method, you can return one or more characters to the stream. Here's the inheritance diagram for the `PushbackReader` class:

```
java.lang.Object
  |____java.io.Reader
        |____java.io.FilterReader
              |____java.io.PushbackReader
```

You'll find the constructors of the `PushbackReader` class in Table 18.62 and its methods in Table 18.63:

Table 18.62: Constructors of the PushbackReader class

Constructor	Does this
`PushbackReader(Reader in)`	It creates a new pushback reader.
`PushbackReader(Reader in, int size)`	It creates a new pushback reader with a pushback buffer of the given size.

Table 18.63: Methods of the PushbackReader class

Method	Does this
`void close()`	It closes the stream.
`void mark(int readAheadLimit)`	It marks the present position in the stream.

Table 18.63: Methods of the PushbackReader class	
Method	**Does this**
`boolean markSupported()`	It returns True if this stream supports the mark operation.
`int read()`	It reads a single character.
`int read(char[ ] cbuf, int off, int len)`	It reads characters into a portion of an array.
`boolean ready()`	It tells whether this stream is ready to be read.
`void reset()`	It resets the stream.
`long skip(long n)`	It keeps character while reading.
`void unread(char[] cbuf)`	It pushes back an array of characters.
`void unread(char[] cbuf, int off, int len)`	It pushes back a section of an array of characters.
`void unread(int c)`	It pushes back a single character.

In Table 18.62, the first form of constructor creates a buffered stream that allows one character to be pushed back at a time and in the second one, the size of the pushback buffer is passed in size.

Working with the **PrintWriter** Class

The methods of this class are used to print formatted representation of objects to a text output stream. Methods of this class never throw I/O exception. If automatic flushing is enabled, that will take place only when one of the `println()`, `printf()`, or `format()` method is invoked. We can check for some error using the `checkError()` method.

Here is the inheritance diagram for the `PrintWriter` class:

```
java.lang.Object
  |____java.io.Writer
        |____java.io.PrintWriter
```

You'll find the constructors of the `PrintWriter` class in Table 18.64 and its methods in Table 18.65:

Table 18.64: Constructors of the PrintWriter class	
Constructor	**Does this**
`PrintWriter(File file)`	It creates a new `PrintWriter` with a specified file without automatic line flushing.
`PrintWriter(File file, String csn)`	It creates a new `PrintWriter` with a specified file and charset, without automatic line flushing.
`PrintWriter(OutputStream out)`	It creates a new `PrintWriter` with existing `OutputStream` without automatic line flushing.
`PrintWriter (OutputStream out, boolean autoFlush)`	It creates a new `PrintWriter` with existing `OutputStream`.
`PrintWriter(String fileName)`	It creates a new `PrintWriter` without automatic line flushing with a given fine name.
`PrintWriter(String fileName, String csn)`	It creates a new `PrintWriter` without automatic line flushing with a given file name and charset.
`PrintWriter(Writer out)`	It creates a `PrintWriter` without automatic line flushing.
`PrintWriter(Writer out, boolean autoFlush)`	It creates a `PrintWriter`.

The flushing of output buffer, whenever the `println()`, `printf()`, or `format()` method is called, is controlled by the `autoFlush` `parameter`. If this parameter is true, the flushing automatically takes place.

Table 18.65: Methods of the PrintWriter class

Method	Does this
`PrintWriter append(char c)`	It appends the given character to the writer.
`PrintWriter append(CharSequence csq)`	It appends the given character sequence to the writer.
`PrintWriter append(CharSequence csq, int start, int end)`	It appends subsequence of given character sequence to the writer.
`boolean checkError()`	It flushes the stream it not closed and checks its error states.
`void clearError()`	It clears error state of the stream.
`void close()`	It closes stream, returns system resources associated with stream.
`void flush()`	It flushes the stream.
`PrintWriter format(Locale l, String format, Object… args)`	It writes args according to the formatted string and let you specify `locale`.
`PrintWriter format(String format, Object… args)`	It writes args according to the formatted string with default `locale`.
`void print(boolean b)`	It prints a `boolean` value.
`void print(char c)`	It prints a `character`.
`void print(char[] c)`	It prints an `array` of `characters`.
`void print(double d)`	It prints a `double-precision floating-point` number.
`void print(float f)`	It prints a `floating-point` number.
`void print(int i)`	It prints an `integer`.
`void print(long l)`	It prints a `long integer`.
`void print(Object ob)`	It prints an `object`
`void print(String s)`	It prints a `string`
`PrintWriter printf(Locale l, String fmt, Object… args)`	It writes arguments to this writer according to formatted `string` and let you specify `Locale`.
`PrintWriter printf(String fmt, Object… args)`	It writes arguments to this writer according to formatted `string`.
`void println()`	It terminates the current line by writing the line separator string.
`void println(boolean x)`	It prints a `boolean` value and then terminates the line.
`void println(char x)`	It prints a `character` and then terminates the line.
`void println(char[] x)`	It prints an `array` of `characters` and then terminates the line.
`void println(double x)`	It prints a `double-precision floating-point` number and then terminates the line.
`void println(float x)`	It prints a `floating-point` number and then terminates the line.
`void println(int x)`	It prints an `integer` and then terminates the line.
`void println(long x)`	It prints a `long integer` and then terminates the line.

Table 18.65: Methods of the PrintWriter class	
Method	**Does this**
`void println(Object x)`	It prints an `Object` and then terminates the line.
`void println(String x)`	It prints a `String` and then terminates the line.
`protected void setError()`	It indicates that an error has occurred.
`void write(char[] buf)`	It writes an `array` of characters.
`void write(char[] buf, int off, int len)`	It writes a portion of an array of characters.
`void write(int c)`	It writes a single `character`.
`void write(String s)`	It writes a `string`.
`void write(String s, int off, int len)`	It writes a portion of a `string`.

Similarly, we have set, of `println()` methods and `write()` methods of this class. The `format()` method is also supported, which works exactly like the `printf()` method.

Here is an example:

```
import java.io.PrintWriter;
public class PrintWriterDemo
{
  public static void main(String[] args)
  {
        PrintWriter out = new PrintWriter(System.out, true);
        out.println("Dreamtech Press");
  }
}
```

Compile and run the code, and the result is shown in Figure 18.11:

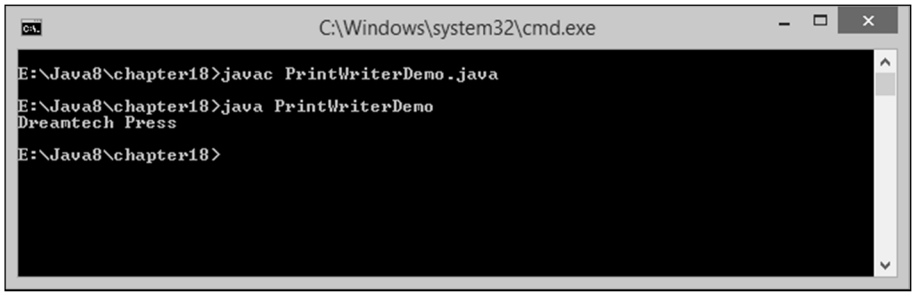


Figure 18.11: Working with the PrintWriter class

Working with the **StreamTokenizer** Class

"What are you doing?" you ask the Novice Programmer. "Writing the code to break a character input stream up into words." "Why not use a stream tokenizer?" you ask. "OK," says the NP, "What's that?"

You can use a `StreamTokenizer` class to break an input stream into tokens, such as words. Here's the inheritance diagram for the `StreamTokenizer` class:

```
java.lang.Object
  |____java.io.StreamTokenizer
```

You'll find the fields of the `StreamTokenizer` class in Table 18.66, its constructors in Table 18.67, and its methods in Table 18.68:

Table 18.66: Fields of the StreamTokenizer class

Field	Does this
`double nval`	The value of a number token.
`String sval`	The string giving the characters of a word token.
`static int TT_EOF`	A constant indicating that end of the stream is read.
`static int TT_EOL`	A constant indicating that the end of the line has been read.
`static int TT_NUMBER`	A constant indicating that a number token has been read.
`static int TT_WORD`	A constant indicating that a word token has been read.
`int ttype`	It contains the type of the token just read.

Table 18.67: Constructors of the StreamTokenizer class

Constructor	Does this

`StreamTokenizer (InputStream is)`	It is deprecated as in JDK 1.1, the preferred way to tokenize an input stream that converts it into a character stream. For example,
	`Reader rd= new BufferedReader(new InputStreamReader(is));`
	`StreamTokenizer strt= new StreamTokenizer(rd);`

`StreamTokenizer(Reader r)`	It constructs a tokenizer that parses the given character-stream.

Table 18.68: Methods of the StreamTokenizer class

Method	Does this
`void commentChar(int ch)`	The character argument starts a single-line comment.
`void eolIsSignificant (boolean flag)`	It returns True if the ends of lines are treated as tokens.
`int lineno()`	It gets the current line number.
`void lowerCaseMode (boolean fl)`	It determines whether word tokens are automatically lowercased.
`int nextToken()`	It parses the next token from the input stream of this tokenizer.
`void ordinaryChar(int ch)`	It specifies that the character argument is ordinary in this tokenizer.
`void ordinaryChars (int low, int hi)`	It specifies that all characters in the range `low` to `high` are considered ordinary in this tokenizer.
`void parseNumbers()`	It specifies that numbers should be parsed by this tokenizer.
`void pushBack()`	It pushes back a token.
`void quoteChar(int ch)`	It indicates that matching pairs of this character delimit string constants.
`void resetSyntax()`	It resets this tokenizer's syntax table so that all characters are "ordinary."
`void slashSlashComments (boolean flag)`	It sets whether the tokenizer recognizes C++-style comments.
`void slashStarComments (boolean flag)`	It sets whether the tokenizer recognizes C-style comments.
`String toString()`	It gets the string representation of the current stream token.

Table 18.68: Methods of the StreamTokenizer class	
Method	**Does this**
`void whitespaceChars (int low, int hi)`	It makes all characters in the range `low` to `high` white space characters.
`void wordChars (int low, int hi)`	It specifies that all characters in the range `low` to `high` are word characters.

Here's an example in which we use `FileReader` to read from a file, `file.txt`, whose content is the string "Welcome to Dreamtech Press". We divide that text into words, verify words with the `TT_WORD` field, and retrieve the current value of token as a string using the `sval` field, which is as follows:

```
import java.io.*;
class StreamTokenizerDemo
{
  public static void main(String args[]) throws Exception
  {
        FileReader filereader = new FileReader("file.txt");
        StreamTokenizer streamtokenizer = new StreamTokenizer(filereader);
        String instring;
        while(streamtokenizer.nextToken() != StreamTokenizer.TT_EOF) {
                if(streamtokenizer.ttype == StreamTokenizer.TT_WORD)
                System.out.println(streamtokenizer.sval); }
        filereader.close(); }
}
```

The output of this example is shown in Figure 18.12 using the `StreamTokenizer` class:

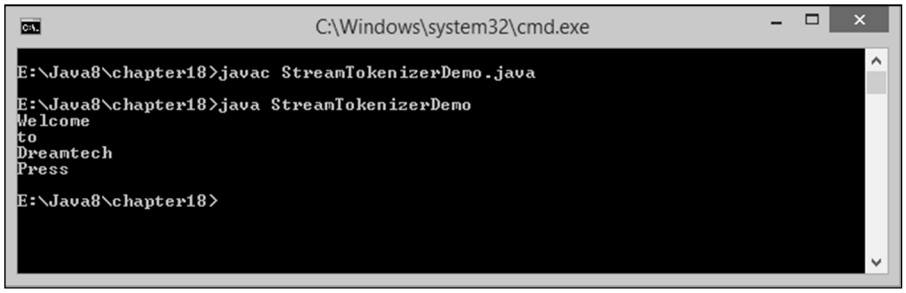


Figure 18.12: Using the StreamTokenizer class

Implementing the **Serializable** Interface

"Uh-oh," says the Novice Programmer, "now I really have a problem. I want to write objects out to a file and read them back in. No way, right?" "Of course there's a way," you say. "You can *serialize* your objects."

Serialization is the process of writing objects to a stream and reading them from the stream. This is particularly useful when you want to save the state of your program to a persistent storage area, such as a file. Later on, you may restore these objects by the process of deserialization. To be serialized, your objects must implement the `Serializable` interface.

This interface has no fields, constructors, or methods—it just shows that an object is serializable. To work with serializable objects, you use the `ObjectInputStream` and `ObjectOutputStream` classes, which are derived from `InputStream` and `OutputStream` respectively. Here's the inheritance diagram for `ObjectInputStream`:

```
java.lang.Object
  |____java.io.InputStream
        |____java.io.ObjectInputStream
```

You'll find the constructors of the `ObjectInputStream` class in Table 18.69 and its methods in Table 18.70:

Table 18.69: Constructors of the ObjectInputStream class

Constructor	Does this
`protected ObjectInputStream()`	It constructs an `ObjectInputStream` object.
`ObjectInputStream (InputStream in)`	It creates an `ObjectInputStream` object that reads from the given InputStream.

Table 18.70: Methods of the ObjectInputStream class

Method	Does this
`int available()`	It gets the number of bytes that can be read.
`void close()`	It closes the input stream.
`void defaultReadObject()`	It reads the nonstatic and nontransient fields of the current class from this stream.
`protected boolean enableResolveObject(boolean enable)`	It enables the stream to allow objects read from the stream to be replaced.
`int read()`	It reads a byte of data.
`int read(byte[] b, int off, int len)`	It reads into an array of bytes.
`boolean readBoolean()`	It reads in a `Boolean`.
`byte readByte()`	It reads an 8-bit `byte`.
`char readChar()`	It reads a 16-bit `char`.
`protected ObjectStreamClass readClassDescriptor()`	It reads a class descriptor from the serialization stream.
`double readDouble()`	It reads a 64-bit `double`.
`ObjectInputStream.GetField readFields()`	It reads the persistent fields from the stream and makes them available by name.
`Float readFloat()`	It reads a 32-bit `float`.
`void readFully(byte[ ] data)`	It reads bytes, blocking until all bytes are read.
`void readFully(byte[] data, int offset, int size)`	It reads bytes, blocking until all bytes are read.
`int readInt()`	It reads a 32-bit `int`.
`String readLine()`	It is deprecated and it uses a buffered stream instead.
`long readLong()`	It reads a 64-bit `long`.
`Object readObject()`	It reads an object from the `ObjectInputStream`.
`protected Object readObjectOverride()`	It is called by trusted subclasses of `ObjectOutputStream`.
`Short readShort()`	It reads a 16-bit `short`.
`Object readUnshared()`	It reads an unshared object from the `ObjectInputStream`.
`protected void readStreamHeader()`	It allows subclasses to read and verify their own stream headers.

Table 18.70: Methods of the ObjectInputStream class	
Method	**Does this**
`int readUnsignedByte()`	It reads an unsigned 8-bit `byte`.
`int readUnsignedShort()`	It reads an unsigned 16-bit `short`.
`String readUTF()`	It reads a UTF format `string`.
`void registerValidation(ObjectInputValidation obj, int prio)`	It registers an object to be validated before the graph is returned.
`protected Class resolveClass(ObjectStreamClass v)`	It loads the local class equivalent of the given stream class description.
`protected Object resolveObject(Object obj)`	It allows trusted subclasses of `ObjectInputStream` to substitute one object for another.
`protected Class resolveProxyClass (String [] interfaces)`	It returns a proxy class that implements the interfaces named in a proxy class descriptor.
`int skipBytes(int len)`	It skips bytes.

Here's the inheritance diagram for `ObjectOutputStream`:

```
java.lang.Object
  |____java.io.OutputStream
        |____java.io.ObjectOutputStream
```

You'll find the constructors of the `ObjectOutputStream` class in Table 18.71 and its methods in Table 18.72.

Table 18.71: Constructors of the ObjectOutputStream class	
Constructor	**Does this**
`protected ObjectOutputStream()`	It creates an `ObjectOutputStream` object.
`ObjectOutputStream(OutputStream out)`	It constructs an `ObjectOutputStream` object that writes to the given OutputStream.

Table 18.72: Methods of the ObjectOutputStream class	
Method	**Does this**
`protected void annotateClass(Class cl)`	It implements this method to allow class data to be stored in the stream.
`protected void annotateProxyClass (Class cl)`	It implements this method to store custom data in the stream along with descriptors for dynamic proxy classes.
`void close()`	It closes the stream.
`void defaultWriteObject()`	It writes the nonstatic and nontransient fields of the current class to this stream.
`protected void drain()`	It clears any buffered data in `ObjectOutputStream`.
`protected boolean enableReplaceObject(boolean enable)`	It enables the stream to replace objects in the stream.
`void flush()`	It flushes the stream.

Table 18.72: Methods of the ObjectOutputStream class

Method	Does this
`ObjectOutputStream.PutField putFields()`	It retrieves the object used to buffer persistent fields.
`protected Object replaceObject(Object obj)`	It allows trusted subclasses of `ObjectOutputStream` to substitute one object for another.
`void reset()`	It disregards the state of any objects already written to the stream.
`void useProtocolVersion(int version)`	It specifies the stream protocol version to use when writing the stream.
`void write(byte[] b)`	It writes an array of bytes.
`void write(byte[ ] b, int off, int len)`	It writes a sub array of bytes.
`void write(int data)`	It writes a `byte`.
`void writeBoolean(boolean data)`	It writes a `Boolean`.
`void writeByte(int data)`	It writes an 8-bit `byte`.
`void writeBytes(String data)`	It writes a string as a sequence of `bytes`.
`void writeChar(int data)`	It writes a 16-bit `char`.
`void writeChars(String data)`	It writes a `string` as a sequence of `chars`.
`void writeDouble(double data)`	It writes a 64-bit double.
`void writeFields()`	It writes the buffered fields to the stream.
`void writeFloat(float data)`	It writes a 32-bit `float`.
`void writeInt(int data)`	It writes a 32-bit `int`.
`void writeLong(long data)`	It writes a 64-bit `long`.
`void writeObject(Object obj)`	It writes the given object to the `ObjectOutputStream`.
`protected void writeObjectOverride(Object obj)`	It is called by trusted subclasses to override the default `writeObject()` method.
`void writeShort(int data)`	It writes a 16-bit `short`.
`protected void writeStreamHeader()`	It is provided so subclasses can append or prepend their own headers to the stream.
`void writeUnshared (Object obj)`	It writes an unshared object to the `ObjectOutputStream`.
`void writeUTF(String data)`	It is primitive data-write operation of the given string.

Now we are back to serialization. Serialization is also required to implement Remote Method Invocation (RMI). The RMI works in this way: RMI on one machine invokes a method of a Java object on a remote machine; an object may be passed as an argument to that remote method; the sending machine serializes the object and transfers it; and finally, the receiving machine deserializes it.

Here's an example showing how serialization works. In this case, we create a new serializable class named `NewString` (file named as `NewString.java`). This class has a constructor that takes a string and one method `toString()`, that returns that string (note that we implement the `Serializable` interface here):

```
import java.io.*;
class NewString implements Serializable {
```

```
    String data;
    public NewString(String instring) { data = instring; }
    public String toString() { return data; }
}
```

After creating a new object of the `NewString` class containing the text "Dreamtech Press!", we use a `FileOutputStream` stream as encapsulated by an `ObjectOutputStream` stream to write that object out to a file—`serialized.dat`. Next, we use a `FileInputStream` stream encapsulated in an `ObjectInputStream` stream to read that object back from the file and display its text. The code is as follows:

```
import java.io.*;
public class SerializationDemo {
  public static void main(String args[]) {
        NewString inobject, outobject;
        inobject = new NewString("");
        outobject = new NewString("Dreamtech Press!");
        try {
                FileOutputStream fileoutputstring = new
                  FileOutputStream("serialized.dat");
                ObjectOutputStream objectoutputstream = new
                  ObjectOutputStream(fileoutputstring);
                objectoutputstream.writeObject(outobject);
                objectoutputstream.flush();
                objectoutputstream.close();
                FileInputStream fileinputstring = new
                FileInputStream("serialized.dat");
                ObjectInputStream objectinputstream = new
                  ObjectInputStream(fileinputstring);
                inobject = (NewString)objectinputstream.readObject();
                objectinputstream.close(); }
        catch(Exception e){ }
        System.out.println(inobject); }
}
```

Compile `NewString.java` and `SerializationDemo.java`. The output of this code is shown in Figure 18.13:

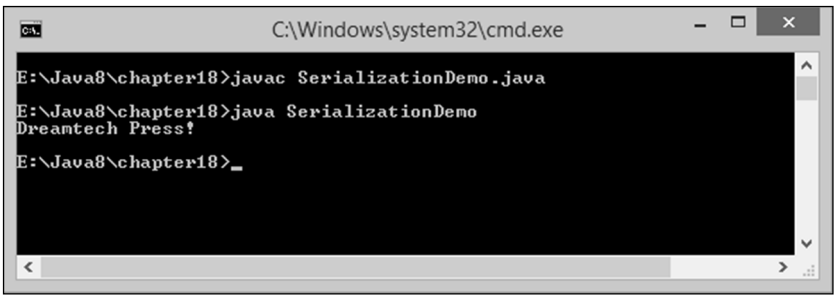


Figure 18.13: Output of the SerializationDemo class

Working with the **Console** Class

The `java.io.Console` class has different methods for accessing character-based console device related to the existing Java Virtual Machine (JVM). The existence of associated console depends upon the manner in which the virtual machine is invoked. If it is started by using the command line, then its console exists, and if it is started by some background job scheduler, then it does not have any console. A connected console of a JVM is represented by the `java.io.Console` class, which can be obtained by using the `System.console()` method. All read and write operations are synchronized. This guarantees the atomic completion of critical operations.

The inheritance diagram of the `Console` class is as follows:

```
java.lang.Object
  |____java.io.Console
```

You'll find the methods of the `Console` class in Table 18.73:

Table 18.73: Methods of the Console class

Method	Does this
`void flush()`	It flushes console and all buffered output is written immediately.
`Console format(String fmt, Object… args)`	It writes a formatted string to this console's output stream according to the given format string and arguments.
`Console printf(String fmt,Object… args)`	It writes a formatted string to this console's output stream using the specified format string and arguments.
`Reader reader()`	It returns unique `Reader` object associated with this console.
`String readLine()`	It reads a single line of text from the console.
`String readLine(String fmt, Object… args)`	It provides a formatted prompt and reads a single line of text from the console.
`char[] readPassword()`	It reads password from the console with echoing disabled.
`char[] readPassword(String fmt, Object… args)`	It provides formatted prompt and reads password from the console with echoing disabled.
`PrintWriter writer()`	It returns unique `PrintWriter` object associated with this console.

With the end of console, the input stream read methods of this class return null. All methods throw `NullPointerException` on passing a null argument to them.

Using this new console API, the formatted output can be written and read easily as it is integrated with Formatter API. The `readLine()` methods are key methods for user input here. `readLine(String fmt, Object… args)` is a combination of formatted output with user input. A first time support to read password from console has been introduced through the `readPassword()` methods.

Here is an example using the `java.io.Console` object:

```
import java.io.*;
public class ConsoleDemo{
  public static void main(String args[]) {
        int MIN_PASSWORD_LENGTH = 8;
        char[] password;
        Console console=System.console();
        do {
                password = console.readPassword( "Enter password (minimum

                of %d characters): ", MIN_PASSWORD_LENGTH);
                console.printf("Password Length is %d\n", password.length);
        } while (password.length < MIN_PASSWORD_LENGTH);
  }
}
```

You can see the result in Figure 18.14, given how the `Console` object is obtained and how its methods are being used:

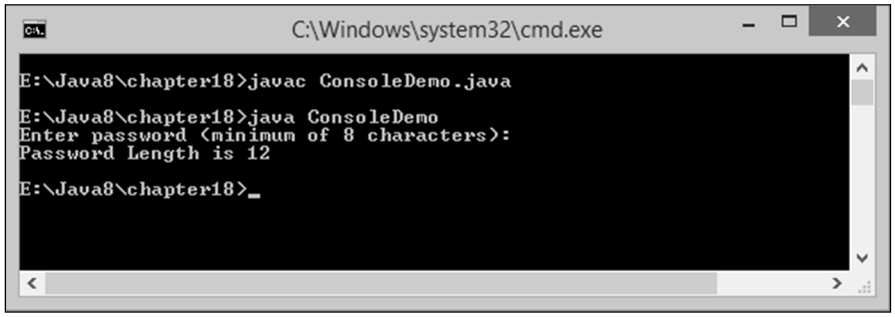


Figure 18.14: Output for the ConsoleDemo class

Execute this code and enter a string of length 8 or more to exit.

Working with the Clipboard

You can use the Java toolkit's `getSystemClipboard()` method and `ClipboardOwner` interface to place data into the Clipboard and retrieve it from the Clipboard.

Here's an example. This code places the text "Dreamtech" into the Clipboard (note that we have to create an object that has a toolkit—a `Frame` object here—to use `getSystemClipboard()`):

```
import java.awt.*;
import java.awt.datatransfer.*;

public class ClipBoardDemo
{
  public static void main(String[] argv)
  {
        useClipboard useclipboard = new useClipboard();
  }
}
class useClipboard extends Frame implements ClipboardOwner
{
  useClipboard()
  {
        Clipboard clipboard = getToolkit().getSystemClipboard();
        StringSelection contents = new StringSelection("Dreamtech");
        clipboard.setContents(contents, this);
        System.exit(0);
  }
  public void lostOwnership(Clipboard clipboard,
  Transferable contents)
  {
        System.out.println("Clipboard contents replaced");
  }
}
```

Running of this code will copy string into the Clipboard. You can paste the copied string "Dreamtech" from the Clipboard to any other editor.

Working with the Printer

You can use the Java toolkit's `getPrintJob()` method to create a new print job for the printer, and you can use the print job's `getGraphics()` method to get a `Graphics` object that you can use to draw on the printed page.

Here's an example that draws a small graphics figure on a page and then prints it (note that we have to create an object that has a toolkit—a `Frame` object here—to use `getPrintJob()`):

```
import java.awt.*;
import java.awt.event.*;

public class PrinterDemo
{
  public static void main(String[] argv)
  {
        usePrinter useprinter = new usePrinter();
  }
}
class usePrinter extends Frame {
  usePrinter() {
        PrintJob printjob = getToolkit().getPrintJob(this,
        "Print graphics", null);
        Graphics g = printjob.getGraphics();
        g.drawRect(2, 2, 100, 100);
        g.drawLine(2, 2, 100, 100);
        g.dispose();
        printjob.end();
        System.exit(0); }
}
```

Being able to use the printer this way is a very powerful aspect of Java I/O. The result is shown in Figure 18.15:

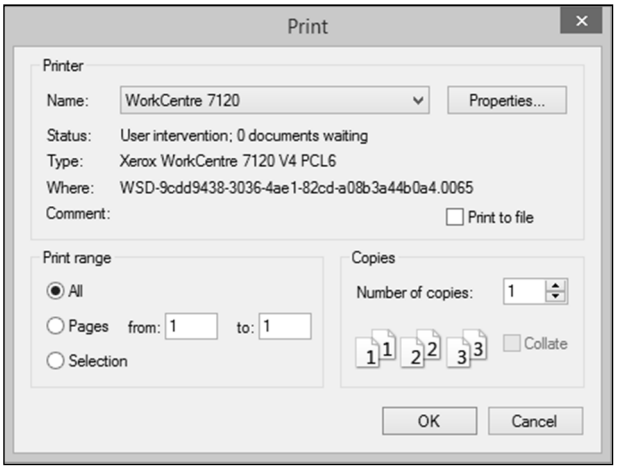


Figure 18.15: Working with the printer

Printing with the **Formatter** Class

Similar to the `printf()` function in C, the class `java.util.Formatter` allows you to perform string formatting. This class was introduced in J2SE 5 and it depends heavily on the varargs feature.

*Using the **System.out.printf()** Method*

The most important thing about `Formatter` is that you should avoid direct interaction with it. Instead of this, you can use the new `PrintStream` method `printf()`. `PrintStream` is not new for you because you are already familiar with `System.out.println()` calls. The following example illustrates the `printf()` method:

```
public class PrinterDemo1
```

```
{
  public static void main(String[] argv)
  {
        String inls = "LKJ";
        String cnt  = "just because";
        System.out.printf("reason: %s (noted by %s)", cnt, inls);
  }
}
```

The output of the preceding code is as follows:

```
reason: just because (noted by LKJ)
```

In the previous `printf()` example, a format string with embedded format specifiers is passed to `printf()`. After this, pass a list of arguments corresponding to each format specifier. The previous example has two format specifiers: `%s` and `%s`, and these correspond to the `cnt` argument and `inls` argument respectively.

The format specifier `%s` has some significance. The letter `s' signifies a `String` conversion and `%` signifies the start of a format specifier. Some characters that trigger the end of format are called *conversions*. A string conversion signifies that Java calls `toString()` on the corresponding argument and substitutes the result for the format specifier.

Using the **String.format()** Method

Sometimes you need only formatted string and print is not required. In such a case, you can employ the `static()` method format on the `String` class. The flollowing example shows some numeric conversions:

```
public class PrinterDemo2
{
  public static void main(String[] argv)
  {
        int i = 91;
        String str = String.format("char: %c integral: %d octal: %o hex:
        %x %n", i, i, i, i);
        System.out.println(str); }
}
```

In the preceding example, platform-specific line separator is denoted by `%n`. After print, the output of `String` would be:

```
char: [  integral: 91 octal: 133 hex: 5b
```

Flagging is supported by numeric conversions that support flags for padding, grouping, justification, and sign.

NOTE

There are four arguments, each using same reference, i.e., you can supply a single argument if all are same and that will be used more than once.

Following example shows how one can use the same argument more than once against a format string:

```
String str = String.format("char: %c integral: %<d octal: %<o hex: %<x %n", i);
```

In the preceding example, reference can be used only once. The less-than sign (<) is used very frequently and bears some special significance. It signifies that the same argument must be used as the last format specifier.

Formatting Dates Using the **String.format()** Method

Many date-related conversions are provided by `Formatter`. For example, consider the following code:

```
import java.util.*;

public class PrinterDemo3
{
  public static void main(String[] argv)
  {
```

```
            int i = 65;
            String str =String.format("%1$td %1$tb %1$ty", new Date());
            System.out.println(str);
      }
   }
```

The preceding code produces a string with the value, as shown in Figure 18.16:

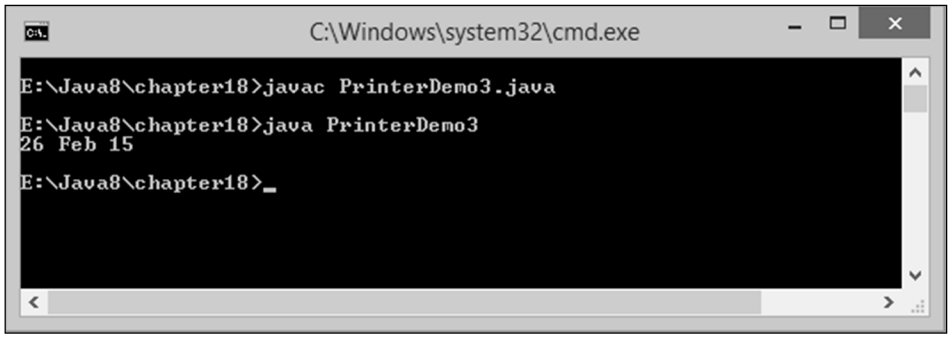


Figure 18.16: Displaying the current date

In the preceding example, all three format specifiers target the first argument. To designate which argument to use, `1$` is used in the format specifier.

Date conversion is done by using a two-character sequence starting with a `t`. The date part to converted, is represented by the second character of the sequence. Consider a sequence `td`. This sequence in the first specifier converts the day of the month.

*Using the **java.util.Formatter** Class*

Nowadays, the `Formatter` class is equipped with many new formatting capabilities. This class supports internationalization by allowing you to pass a Locale to the constructor.

The `Appendable` object is the latest interface that defines the `append()` methods. `Formatter` allows to pass an `Appendable` object. Since the `Appendable` objects encapsulate the `append()` methods, formatter can store its results in a text collector, such as a `stream` object. To implement this interface, all relevant Java classes (e.g., `StringBuffer` and `PrintStream`) are modified by Oracle. The following example shows how to implement it:

```
import java.util.*;

public class PrinterDemo4
{
   public static void main(String[] argv)
   {
         double avg     = 6.0e23;
         StringBuffer buf = new StringBuffer();
         Formatter frmter = new Formatter(buf, Locale.US);
         frmter.format("avg's number: %e %n", avg);
         frmter.format("base of the natural log: %e %n", Math.E);
         System.out.println(buf.toString());
   }
}
```

The corresponding output is shown in Figure 18.17 doing the calculations:

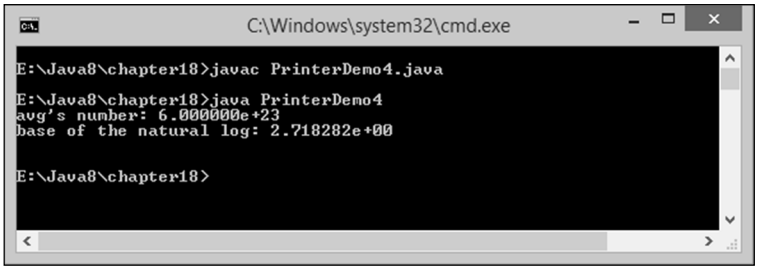


Figure 18.17: Performing calculation

The `Formatter` class encapsulates many conversions and some additional features. See Java API documentation pages for `java.util.Formatter` for more details.

Scanning Input with the **Scanner** class

You can read input from either the keyboard or a text file by using the `Scanner` class. This class provides simple and Sun-standard mechanism for reading input. When you are using `tjXXXXx.jar` file, you can access the `Scanner` class by importing `edu.fcps.Scanner`, and after J2SE 5, you can access the complete `Scanner` class by importing `java.util.Scanner`. A simple text Scanner is used for parsing primitive types and strings through regular expressions.

Whenever a Scanner receives an input, it breaks it into tokens. For breaking, Scanner uses delimiter pattern, which by default matches whitespace. These tokens are then converted into values of different types using the methods discussed here. The following code allows a user to read a number from `System.in`:

```
Scanner scan = new Scanner(System.in);
int i = scan.nextInt();
```

Consider another example in which code allows long types to be assigned from entries in a file `myNumbers`:

```
Scanner scan = new Scanner(new File("myNumbers"));
while (scan.hasNextLong()) { long aLng = scan.nextLong(); }
```

We know that the default value of delimiters is whitespace, but Scanner also allows the use of delimiter other than whitespace. The following example reads several items from a string:

```
import java.util.*;
public class ScannerDemo{
  public static void main(String[] argv) {
        String input = "10 fish 20 fish green fish white fish";
        Scanner s = new  Scanner(input).useDelimiter("\\s*fish\\s*");
        System.out.println(s.nextInt());
        System.out.println(s.nextInt());
        System.out.println(s.next());
        System.out.println(s.next());
        s.close(); }
}
```

The output of the code is shown in Figure 18.18, showing the text in different lines:

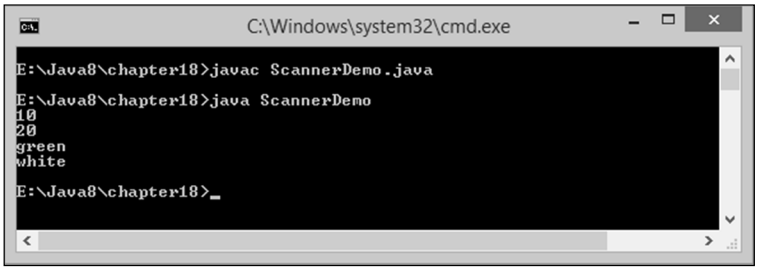


Figure 18.18: Scanning input with the ScannerDemo class

The output of the preceding code is shown in Figure 18.18, taking input from the Scanner class.

Summary

In this chapter, you have learned the use of file class and to work with streams. Then, we have explored the RandomAccessFile, Reader and Writer classes. Following that, we have discussed to input the value from the keyboard with the InputStreamReader class and work with the OutputStreamWriter class. You have next learned to work with files, character arrays, and buffers. Moving further, we have discussed how to work with the StreamTokenizer class, Serializable interface, and Console class. You have also learned to work with printer and clipboard. Lastly, you have learned to print with the Formatter class and scan input with the Scanner class.

In the next chapter, you will learn to work with multiple threads.



19

Working with Multiple Threads

If you need an immediate solution to:	*See page:*
Getting the Main Thread	678
Naming a Thread	678
Pausing a Thread	679
Creating a Thread with the Runnable Interface	681
Creating a Thread with the Thread Class	684
Creating Multiple Threads	686
Joining Threads	688
Checking if a Thread Is Alive	689
Setting Thread Priority and Stopping Threads	690
Synchronizing	692
Communicating between Threads	704
Suspending and Resuming Threads	705
Creating Graphics Animation with Threads	707
Eliminating Flicker in Graphics Animation Created Using Threads	710
Suspending and Resuming Graphics Animation	711
Using Double Buffering	713
Simplifying Producer-Consumer with the Queue Interface	715
Implementing Concurrent Programming	717
Simplifying Servers Using the Concurrency Utilities	718
Knowing Various Concurrency Utilities	719
Learning about the java.util.concurrent Package	719
Learning about the java.util.concurrent.locks Package	721
Learning about the java.util.concurrent.atomic Package	722

In Depth

In this chapter, we are going to take a look at multithreaded programming in Java and how it lets you support graphics animation. A *thread* is a lightweight entity within a process. Multiple threads can occur within the same process and share the memory. In fact, Java is one of the few programming languages that supports multithreading. We can also say that multithreading is a specialized form of multitasking.

There are two types of multitasking—*Process-based* and *Thread-based*. In process-based multitasking, the computer is able to run two or more programs concurrently. For example, you can listen to music and use the notepad at the same time. On the other hand, a thread-based multitasking involves a single program to perform two or more tasks simultaneously. For example, a text editor can format text at the same time it is being printed.

With the use of multithreading, maximum usage of resources occurs in minimum time. By keeping the idle time to minimum, efficient programs can be created that makes maximum utilization of the CPU.

NOTE

Many components in Java, most Swing component methods notably, are not thread-safe.

Using Threads in Java

When a Java program starts, it has one thread—the main thread. You can interact with the main thread in various ways (as you'll see in this chapter) such as getting or setting its name, pausing it, and more. However, you can also start other threads. You call the `start()` method of the object to start the thread and place the code that you want to execute in the `run()` method. There are two ways to create new threads. One is to declare a class that is a subclass of the Thread class. This subclass should override the run method of the Thread class, and you can then allocate and start an instance of that class. For example, here's how you can define a new thread class that starts when you create an instance of it:

```
class CustomThread extends Thread {
CustomThread() { start(); }
public void run() { } //Do the work this thread was created for
}
```

You can then create and start a thread of this type, like this:

```
CustomThread thread1 = new CustomThread();
```

Another way to create a thread is to create a class that implements the Runnable interface.

Secondly, this class then implements the `run()` method. An instance of the class that is passed at the time of creating a thread object can then be allocated. Here's how this looks in code:

```
class CustomThread implements Runnable {
Thread thread;
CustomThread() {
thread = new Thread(this, "second");
thread.start(); }
public void run() { } //Do the work this thread was created for
}
```

This code would then create a thread of this type and start it running:

```
CustomThread thread1 = new CustomThread();
```

It's also worth noting that every thread has a name for identification purposes (in fact, more than one thread may have the same name). Every thread also has a priority, and threads with higher priorities are executed in preference to threads with lower priorities.

Let us take one more example to understand this better.

```
class CreateThreadDemo {
  public static void main(String args[]) {
        new NewThread();        // create a new thread
```

```
            try {
                    for(int k = 10; k > 6; k--) {
                            System.out.println("Parent Thread: " + k);
                            Thread.sleep(1000); }
            }
            catch (InterruptedException e) {
                    System.out.println("Parent Interrupted."); }
            System.out.println("Exiting Parent."); }
    }
    class NewThread implements Runnable{
      Thread T;
      NewThread() { //a new, second thread
            T = new Thread(this, "Creating Thread");
            System.out.println("Child Thread : " + T);
            T.start(); }            // Start the thread
      // Point of entry for the second thread.
      public void run() {
            try {
                    for(int j = 10; j > 6; j--) {
                            System.out.println("Child Thread: " + j);
                            Thread.sleep(500); } }
            catch (InterruptedException e) {
                    System.out.println("Child interrupted."); }
            System.out.println("Exiting Child."); }
    }
```

Here's the output of the code:

```
Child Thread: Thread[Creating Thread,5,main]
Parent Thread: 10
Child Thread:10
Child Thread: 9
Child Thread:8
Parent Thread: 9
Child Thread:7
Exiting Child.
Parent Thread: 8
Parent Thread: 7
Exiting Parent.
```

Life Cycle of a Thread

A thread is the smallest unit of processing. It is lightweight and therefore easy to access. From the creation of the thread to its termination, it follows a cycle known as the life cycle of a thread. According to Oracle, a thread has six states. They are as follows:

- New
- Runnable
- Blocked
- Waiting
- Timed_Waiting
- Terminated

The diagrammatical representation of the life cycle of a thread is shown in Figure 19.1:

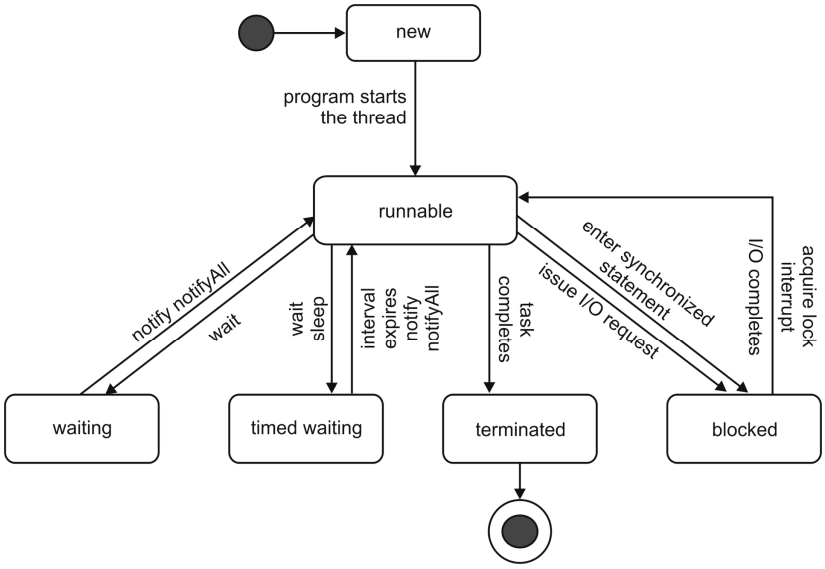


Figure 19.1: Thread Life Cycle UML State Diagram

The various states in the life cycle of a thread, as the figure shows, are as follows:

1. **New**—Before the `start()` method is invoked and an instance of a thread is created, the thread comes to new state.
2. **Runnable**—The life of a thread begins from this state. After the `start()` method is invoked, the thread first enters the runnable state. It can also enter this state after running, waiting, sleeping, or coming back from the blocked state. Here, the thread is waiting for its request to be addressed by the processor.
3. **Blocked**—When the thread is in the waiting or suspended or blocked state, it is not allowed to enter the runnable state or running state; then it appears to be in the blocked state.
4. **Waiting**—The thread is said to be in the waiting state when it waits indefinitely for another thread to complete an action.
5. **Timed_Waiting**—A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state. When the thread is waiting for a specified time interval and transitions back to the runnable state on expiration of the interval or occurrence of what it was waiting for, it is said to be in the `Timed_Waiting` state.
6. **Terminated**—When the thread has completed the execution of the `run()` method, its life cycle comes to an end. The thread is killed by invoking the `stop()` method, coming to the dead state.

Synchronization of Threads

Thread synchronization, also known as serialization, is a mechanism to ensure that no two concurrently executing threads possess the same parts of a program at the same time. If a thread holds some part of a process to access, the other thread must wait till the execution of the first thread finishes.

Basically, the object accessed by a thread is locked by it so that no other thread can access it. Synchronization is used to control access to the systems, such as multiprocessor systems, systems supporting multithreading, and distributed systems, consisting of bulky data, such as database for colleges or hospitals and many more.

There are two synchronization techniques:

- To control the order in which the code is executed to perform different tasks
- To prevent the obstacles that can arise when two threads share the same resource at the same time

Multithreaded Custom Class Loader

The function of the `java.lang.ClassLoader` class is to locate the bytecode for some specific class and then transform that bytecode into a usable class during runtime. The runtime system provides class loaders that can locate bootstrap classes, extension classes, and user classes.

If custom class loaders restrict to an acyclic class loader delegation model, they will never face the condition of deadlocks. The ClassLoader recognizes the acyclic delegation model. In this type of model, a class loader always consists of a parent (delegate). When a request for a class is made, the primary task of the class loader is to first check whether the class requested is already loaded. If the class is not already found, the class loader asks its parent to find the location of the class; and if the parent is not able to locate the class either, the class loader itself tries to find the location of the class.

In Java 8, the process of creation of new custom loaders is similar to those in the previous versions of Java, that is, create a subclass of ClassLoader and then override the `findClass()` and `loadClass()` methods.

With the release of Java SE, you can avoid deadlocks by following these rules:

- Ensure that the custom class loader for the concurrent class loading is safe.
  a. Decide for the internal locking scheme. Example: `java.lang.ClassLoader` uses the locking scheme based on the name of class that is requested.
  b. Remove all synchronization on the lock of the class loader object.
  c. Ensure that critical sections are protected in case of multiple threads loading different classes.
- Invoke java.lang.ClassLoader's static method `registerAsParallelCapable()`. This signifies that all the instances of the custom class loader are safe.
- Ensure that all the classes that the custom class loader extends also invoke the `registerAsParallelCapable()` method in their class initializers. Ensure that they are multithread safe for concurrent class loading.

That's it for the overview of what's in this chapter. It's time to turn to the "Immediate Solutions" section.

Immediate Solutions

Getting the Main Thread

The NP appears and says, "Threads? I have been programming Java for a while now, and I can't see any threads." You smile and say, "Every Java program has at least one thread, called the *main* thread, and you can take a look at it with the `currentThread()` method."

The `currentThread()` method of the `Thread` class gets the current thread; if you are executing in the main thread, `this` method gets the main thread. Here's an example in which we are getting the main thread of an application and printing out its name by using the `getName()` method (note that you can set a thread's name with `setName()`—see the next solution):

```
class MainThread
{
  public static void main(String args[])
  {
        Thread thread = Thread.currentThread();
        System.out.println("The name of main thread is  \"" + thread.getName()+"\"");
  }
}
```

The output of this code is shown in Figure 19.2, displaying the name of the main thread:

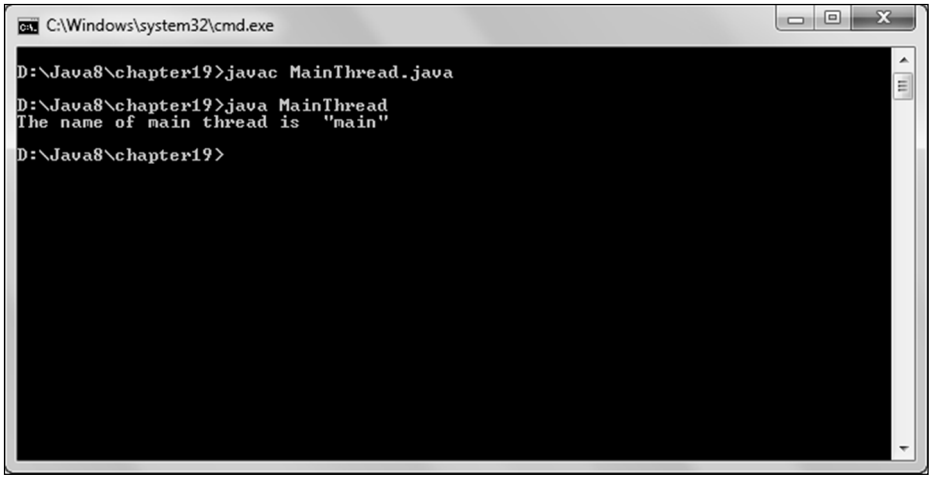


Figure 19.2: Displaying the Name of the Main Thread

Naming a Thread

One way to distinguish threads is by name, and you can set the name of a thread with the `setName()` method. Here's an example in which we rename the main thread in an application and display that name:

```
class SetThreadName
{
  public static void main(String args[])
  {
        Thread thread = Thread.currentThread();
        System.out.println("Main thread's original name is " +
        thread.getName());
        thread.setName("Kogent");
```

```
            System.out.println("Main thread's name is now " + thread.getName());
        }
    }
```

The output of this code is shown in Figure 19.3, changing the name of the thread:

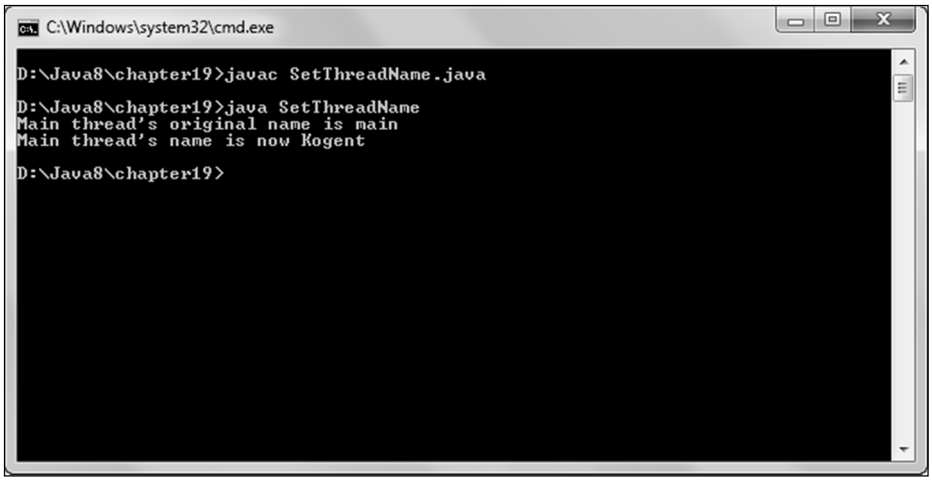


Figure 19.3: Giving Name to the Main Thread

After you have given a thread a name, you can use the `getName()` method to check the name and thus; determine which thread you are working with in a section of code that many threads execute (such as the code in a `Thread` class's `run()` method).

Pausing a Thread

Sometimes, you want to pause a thread's execution, and you can do that with the `sleep()` method. You pass this method the amount of time to pause, in milliseconds (a millisecond is *one-thousandth* of a second), and the thread will wait that amount of time before continuing. Here's an example in which we print out a message, pausing for a second between words:

```
class SleepDemo
{
  public static void main(String args[])
  {
        try
        {
                System.out.println("Welcome");
                Thread.sleep(1000);
                System.out.println("to");
                Thread.sleep(1000);
                System.out.println("Kogent Solutions.");
        }
        catch (InterruptedException e) { }
  }
}
```

The result is shown in Figure 19.4:

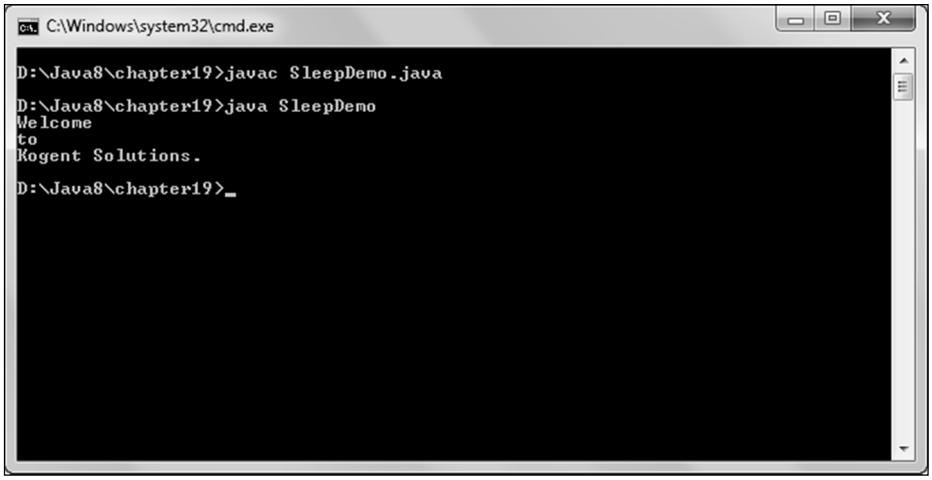


Figure 19.4: Pausing a Thread

When you run this code, you'll see that the words appear with a one-second pause between them.

Let's look at another example on sleeping in Figure 19.5:

```
class NewThread implements Runnable {
  String Tn;
  Thread t;
  NewThread(String threadname) {
        Tn = threadname;
        t = new Thread(this, Tn);
        System.out.println("New thread: " + t);
        t.start(); }
  public void run() {
        try {
                for(int j = 75; j > 50; j--) {
                        System.out.println(Tn + ": " + j);
                        Thread.sleep(100); }
        }
        catch (InterruptedException e) {
                System.out.println(Tn + " breaking up..."); }
        System.out.println(Tn + " finishing..."); }
}
class Sleep {
  public static void main(String args[]) {
        NewThread ob1 = new NewThread("1");
        NewThread ob2 = new NewThread("2");
        try {
                Thread.sleep(1500);
                ob1.t.suspend();
                System.out.println("Thread 1's sleeping in progress");
                Thread.sleep(1500);
                ob1.t.resume();
                System.out.println("Waking up thread One");
                ob2.t.suspend();
                System.out.println("Thread 2 starting to sleep...");
```

```
                Thread.sleep(1500);
                ob2.t.resume();
                System.out.println("Thread Two, waking up..."); }
        catch (InterruptedException e) {
                System.out.println("Main thread broken"); }
        try {
                System.out.println("Waiting for threads to finish.");
                ob1.t.join();
                ob2.t.join(); }
        catch (InterruptedException e) {
                System.out.println("Main thread broken"); }
        System.out.println("Finishing Main thread"); }
}
```



Figure 19.5: Sleeping of a Thread

Creating a Thread with the **Runnable** Interface

"I want to create a new thread to spell check a document while the user is doing other things," the NP says. "How do I do that?" "Well," you say, "there are two ways to create your own threads—implementing the `Runnable` interface and extending the `Thread` class." "Hmm," says the NP, "let's start with `Runnable`."

Implementing the `Runnable` interface is one of the two ways to create your own threads in Java. This interface has only one method: `run()`. You place the code you want to have the new thread execute in that method. When you create a new `Runnable` object, you can pass that object to the constructor of the `Thread` class to create the new thread.

Here's an example in which we create a new class that implements the `Runnable` interface. It passes itself to a `Thread` class constructor and starts itself when an object of this class is created. When it runs, it prints out its name once a second for 10 seconds and then quits. Here's the code for this new `Runnable` class, which we call `SecondThread` (note that the actual work is done in the `run()` method and that the code is executed with the `Thread` class's `start()` method):

```
class SecondThread implements Runnable {
  Thread thread;
  SecondThread() {
        thread = new Thread(this, "second");
        System.out.println("Starting Child Thread.");
        thread.start(); }
  public void run() {
```

```
            try {
                    (Thread.currentThread()).setName("Child");
                    for(int loop_index = 0; loop_index < 10; loop_index++) {
                    System.out.println((Thread.currentThread()).getName()+
                    " thread here...\n");
                    Thread.sleep(1000); }
            }
            catch (InterruptedException e) { }
            System.out.println("Second thread Ending."); }
    }
```

Here's the application that will create a new thread of the `SecondThread` class and run it; note that we are also having the main thread print out its name so that you can see the two threads alternatively:

```
    class RunnableDemo {
      public static void main(String args[]) {
            System.out.println("Starting Parent Thread.");
            SecondThread secondthread = new SecondThread();
            try {
                    (Thread.currentThread()).setName("Parent");
                    for(int loop_index = 0; loop_index < 10; loop_index++) {
                    System.out.println((Thread.currentThread()).getName()+
                    " thread here...");
                    Thread.sleep(1000); }
            }
            catch (InterruptedException e) { }
            System.out.println("Parent Thread Ending."); }
    }
```

The result of this code is shown in Figure 19.6:

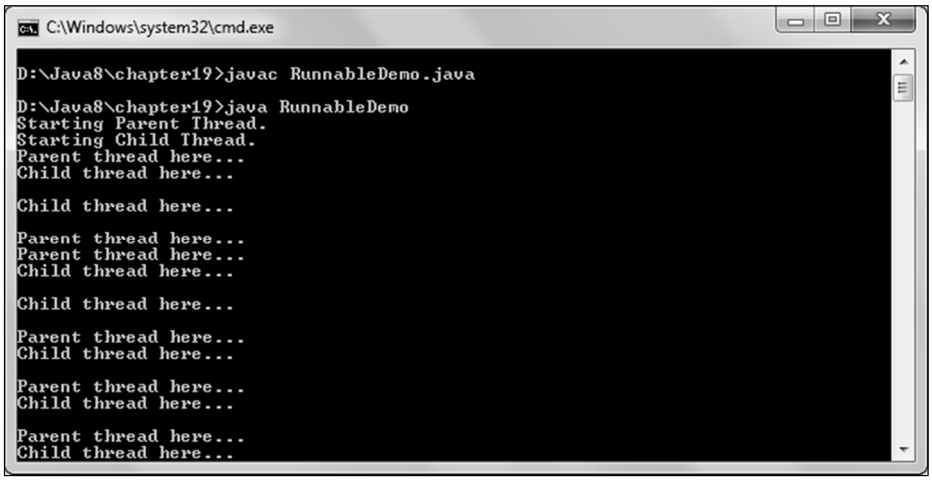


Figure 19.6: Alternate Printing of Threads

Note that the two threads alternately print out their names (also note that you can't guarantee that they'll be displayed alternatively alternate—that's up to the operating system).

Let us take another example of creating a thread. The easiest way to do that is to write a class that extends the `Thread` class. All that needed to start a thread is to create an instance of the `Thread` class and call its `start()` method.

Look at this class named `CountDownClock` clock that extends the `Thread` class:

```
    public class CountDownClock extends Thread
    {
      public void run()
      {
            for (int t = 20; t >= 0; t--)
            {
```

```
                    System.out.println("T minus " + t);
                    try { Thread.sleep(1000); }
                    catch (InterruptedException e) { }
            }
      }
}
```

Now, let us create a class that extends the `Thread` class. But, there is a twist—the `Thread` class can have one superclass. In this case, you must create a class that implements the `Runnable` interface instead of extending the `Thread` class.

As you already know that the `Runnable` interface marks an object that can be run as a thread. So, a code for the creation of the `Runnable` object named `LaunchTask` is as follows:

```
public class LaunchTask implements Runnable
{
  private int start;
  private String message;
  public LaunchTask(int start, String message)
  {
        this.start = start;
        this.message = message;
  }
  public void run()
  {
        try { Thread.sleep(20000 - (start * 1000)); }
        catch (InterruptedException e){ }
        System.out.println(message);
  }
}
```

Now, we'll write the code for the `main()` method that will use `LaunchTask` and `CountDownClock` to produce the result:

```
public class CountDown
{
  public static void main(String[] args)
  {
        Thread clock = new CountDownClock();
        Runnable flood, ignition, liftoff;
        flood = new LaunchTask(16, "READY->Flood the pad!");
        ignition = new LaunchTask(6, "STEADY-->Start engines!");
        liftoff = new LaunchTask(0, "GO-->Liftoff!");
        clock.start();
        new Thread(flood).start();
        new Thread(ignition).start();
        new Thread(liftoff).start();
  }
}
```

Here's the result of the code:

```
D:\Java8\chapter19>java CountDown
T minus 20
T minus 19
T minus 18
T minus 17
T minus 16
READY→Flood the pad!
T minus 15
T minus 14
T minus 13
T minus 12
T minus 11
T minus 10
T minus 9
T minus 8
T minus 7
T minus 6
```

```
STEADY→Start engine!
T minus 5
T minus 4
T minus 3
T minus 2
T minus 1
GO→Liftoff!
T minus 0
```

Creating a Thread with the **Thread** Class

"OK," says the NP, "I know I can create a new thread by using the `Runnable` interface..." "Or," you say, "using the `Thread` class." "Tell me more!" says the NP.

The `Thread` class supports new threads. It also implements the `Runnable` interface, so you don't have to. Here's the inheritance diagram for `Thread`:

```
java.lang.Object
  |____java.lang.Thread
```

You'll find the fields for the `Thread` class in Table 19.1, its constructors in Table 19.2, and its methods in Table 19.3:

Table 19.1: Fields of the Thread class

Field	Does this
`static int MAX_PRIORITY`	The maximum priority a thread can have
`static int MIN_PRIORITY`	The minimum priority a thread can have
`static int NORM_PRIORITY`	The default priority assigned to a thread

Table 19.2: Constructors of the Thread class

Constructor	Does this
`Thread()`	It constructs a new `Thread` object.
`Thread(Runnable target)`	It constructs a new `Thread` object using a `Runnable` object.
`Thread(Runnable target, String name)`	It constructs a new `Thread` object with a `Runnable` object and a name.
`Thread(String name)`	It constructs a new `Thread` object with a name.
`Thread(ThreadGroup group, Runnable target)`	It constructs a new `Thread` object as part of a group.
`Thread(ThreadGroup group, Runnable target, String name)`	It constructs a new `Thread` object with a `Runnable` object. This `Thread` object has the specified name as its name and belongs to the thread group referred to by `group`.
`Thread(ThreadGroup group, Runnable target, String name, long stackSize)`	It constructs a new `Thread` object so that it has target as its run object, has the specified name as its name, belongs to the thread group referred to by group, and has the specified stack *size*.
`Thread(ThreadGroup group, String name)`	It constructs a new `Thread` object that's part of a group and has a name.

Table 19.3: Methods of the Thread class

Method	Does this
`static int activeCount()`	It gets the current number of active threads in this thread's thread group.
`void checkAccess()`	It determines whether the currently running thread has the permission to modify this thread.
`int countStackFrames()`	It is deprecated. The definition of this call depends on `suspend`, which is deprecated.
`static Thread currentThread()`	It gets a reference to the currently executing `Thread` object.

Table 19.3: Methods of the Thread class

Method	Does this
`static void dumpStack()`	It prints a stack trace of the current thread.
`static int enumerate(Thread[] tarray)`	It copies to the indicated array every active thread in this thread's thread group.
`ClassLoader getContextClassLoader()`	It gets the context `ClassLoader` for this thread.
`String getName()`	It gets this thread's name.
`int getPriority()`	It gets this thread's priority.
`ThreadGroup getThreadGroup()`	It gets the thread group to which this thread belongs.
`void interrupt()`	It interrupts this thread.
`static boolean interrupted()`	It returns true if the current thread has been interrupted.
`boolean isAlive()`	It returns true if this thread is alive.
`boolean isDaemon()`	It returns true if this thread is a daemon thread.
`boolean isInterrupted()`	It returns true if this thread has been interrupted.
`void join()`	It waits for this thread to die.
`void join(long millis)`	It waits at most `millis` milliseconds for this thread to die.
`void join(long millis, int nanos)`	It waits at most `millis` milliseconds plus `nanos` nanoseconds for this thread to die.
`void resume()`	It is deprecated. This method exists solely for use with suspend, which is deprecated.
`void run()`	If this thread was constructed using a separate `Runnable` interface `run` object, that `Runnable` object's `run` method is called.
`void setContextClassLoader (ClassLoader cl)`	It sets the context `ClassLoader` for this thread.
`void setDaemon(boolean on)`	It marks this thread as either a daemon thread or a user thread.
`void setName(String name)`	It changes the name of this thread to be equal to the argument name.
`void setPriority (int newPriority)`	It changes the priority of this thread.
`static void sleep (long millis)`	It causes the currently executing thread to sleep for the indicated number of milliseconds.
`static void sleep (long millis, int nanos)`	It causes the currently executing thread to sleep for the indicated number of milliseconds plus the indicated number of nanoseconds.
`void start()`	It causes this thread to begin execution, which means that the Java Virtual Machine will call the `run` method of this thread.
`void stop()`	It is deprecated. This method is considered inherently unsafe.
`void stop(Throwable obj)`	It is deprecated. This method is considered inherently unsafe.
`void suspend()`	It is deprecated. This method has been deprecated because it's considered inherently deadlock prone.
`String toString()`	It gets a string representation of this thread.
`static void yield()`	It indicates that the current thread is willing to yield its current use of a processor.

Here's an example in which we create a thread and have it print out its name 10 times, once a second. Note that in this case, we are calling the thread's constructor and passing it a name for this thread, and then we are calling the thread's `start()` method to start execution of the code in the `run` method. The action takes place in the `run()` method, that's where we have the thread print out its name every second. Here's the code:

```
class ChildThread extends Thread { ChildThread() { super("Child"); start(); }
  public void run() { try {
                System.out.println("Child Thread Started.");
                for(int loop_index = 0; loop_index < 10; loop_index++) {
                System.out.println((Thread.currentThread()).getName()+ "
                thread here...");
                Thread.sleep(1000);
                }
        }
        catch (InterruptedException e) { }
        System.out.println("Child Thread Ended."); }
}
```

Here's an application in which we use the `ChildThread` class and also print out the name of the main thread every second:

```
class ThreadDemo {
  public static void main(String args[]) {
        ChildThread childthread = new ChildThread();
        try {
                for(int loop_index = 0; loop_index < 10; loop_index++) {
                System.out.println((Thread.currentThread()).getName()+
                " thread here...");
                Thread.sleep(1000); }
        }
        catch (InterruptedException e) { }
  }
}
```

The result of this code is shown below. Note that both threads are running and displaying their names, once a second:

```
D:\Java8\chapter19>java ThreadDemo
main thread here...
Child Thread Started.
Child thread here...
main thread here...
Child thread here...
main thread here...
Child thread here...
main thread here...
Child thread here...
main thread here...
Child thread here...
main thread here...
Child thread here...
main thread here...
Child thread here...
main thread here...
Child thread here...
main thread here...
Child thread here...
main thread here...
Child thread here...
Child Thread Ended.
```

Creating Multiple Threads

"I know I can start a second thread," the NP says, "but what about starting *four* new threads?" You smile and say, "No problem."

You can create and run multiple threads in a program—just give each thread a new object. Here's an example in which we create four new threads and have each one print out its name once a second:

```
class CustomThread extends Thread {
  CustomThread(String name) {
        super(name);
        start(); }
```

```
    public void run() {
          try {
          for(int loop_index = 0; loop_index < 4; loop_index++) {
                System.out.println((Thread.
                currentThread()).getName()+ " thread here...");
                Thread.sleep(1000); }
    }
    catch (InterruptedException e) { }
    System.out.println((Thread.currentThread()).getName() +" ending."); }
  }
  class MultiThreadDemo {
    public static void main(String args[]) {
          CustomThread thread1 = new CustomThread("First");
          CustomThread thread2 = new CustomThread("Second");
          CustomThread thread3 = new CustomThread("Third");
          CustomThread thread4 = new CustomThread("Fourth");
          try {
                for(int loop_index = 0; loop_index < 10; loop_index++) {
                System.out.println((Thread.currentThread()).getName()+ "
                thread here...");
                Thread.sleep(1000); }
          }
          catch (InterruptedException e) { }
    }
  }
```

The result of this code is shown below:

```
D:\Java8\chapter19>java MultiThreadDemo
First thread here...
Third thread here...
Second thread here...
main thread here...
Fourth thread here...
Second thread here...
First thread here...
Third thread here...
main thread here...
Fourth thread here...
Second thread here...
Third thread here...
First thread here...
main thread here...
Fourth thread here...
Third thread here...
First thread here...
Second thread here...
main thread here...
Fourth thread here...
First ending.
Second ending.
Third ending.
main thread here...
Fourth ending.
main thread here...
main thread here...
main thread here...
main thread here...
main thread here...
```

Let us take another example to understand this concept better:

```
class NewThread implements Runnable {
  String name;
  Thread t;
  NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
```

```
            System.out.println("New thread: " + t);
            t.start(); }
      public void run() {
            try {
                  for(int i = 100; i > 95; i--) {
                        System.out.println(name + ": " + i);
                        Thread.sleep(1000); }
            }
            catch (InterruptedException e) {
                  System.out.println(name + "Interrupted"); }
            System.out.println(name + " exiting."); }
   }
   class StartThread {
      public static void main(String args[]) {
            new NewThread("1:");
            new NewThread("2:");
            new NewThread("3:");
            try { Thread.sleep(10000); }
            catch (InterruptedException e) {
                  System.out.println("Parent thread Interrupted"); }
            System.out.println("Parent thread exiting."); }
   }
```

The result of the code is shown below:

```
New thread: Thread[1:,5,main]
New thread: Thread[2:,5,main]
New thread: Thread[3:,5,main]
1:: 100
3:: 100
2:: 100
2:: 99
3:: 99
1:: 99
2:: 98
1:: 98
3:: 98
2:: 97
3:: 97
1:: 97
2:: 96
1:: 96
3:: 96
2: exiting.
3: exiting.
1: exiting.
Parent thread exiting.
```

Joining Threads

"I need more control over threads," the NP says. "Say I need to be able to wait until a thread is done executing before going on with the rest of the program. Any way to check that?" "Certainly." you say. "You can use the `join()` method to wait until a thread is done." "Really?" the NP asks excitedly.

The `Thread` class's `join()` method waits until a thread is finished executing (also called *waiting for a thread to die*) before returning. Here's an example in which we create four new threads and wait for each one to finish executing before ending the main application:

```
class CustomThread extends Thread
{
  CustomThread(String name)
  {
        super(name);
        start();
  }
  public void run()
  {
```

```
            try {
                    for(int loop_index = 0; loop_index < 4; loop_index++) {
                    System.out.println((Thread.currentThread()).getName()+ "
                    thread here...");
                    Thread.sleep(1000); } }
            catch (InterruptedException e) { }
            System.out.println((Thread.currentThread()).getName()+" ending.");}
      }
      class JoinDemo {
            public static void main(String args[]) {
                    CustomThread thread1 = new CustomThread("First");
                    CustomThread thread2 = new CustomThread("Second");
                    CustomThread thread3 = new CustomThread("Third");
                    CustomThread thread4 = new CustomThread("Fourth");
                    try {
                            thread1.join();
                            thread2.join();
                            thread3.join();
                            thread3.join(); }
                    catch (InterruptedException e) { }
            }
      }
```

The output of this code is shown below:

```
C:\src>java JoinDemo
Second thread here...
First thread here...
Third thread here...
Fourth thread here...
Fourth thread here...
Third thread here...
Second thread here...
First thread here...
Third thread here...
First thread here...
Second thread here...
Fourth thread here...
Second thread here...
First thread here...
Fourth thread here...
Third thread here...
First ending.
Third ending.
Fourth ending.
Second ending.
```

Checking if a **Thread** Is Alive

"Uh-oh," says the NP, "I have lost a thread. How do I know if it's still doing anything?" "No problem at all!" you say. "To check on a thread, you can use the `isAlive()` method to see whether the thread is still active." "How does it work?" the NP asks.

Here's an example in which we create and run four new threads. As the threads are executing, we use the `isAlive()` method on the first new thread, and when all threads are done (which we verify with the `join()` method), we check that thread again with `isAlive()`. Here's the code:

```
class CustomThread extends Thread
{
  CustomThread(String name)
  {
        super(name);
        start(); }
  public void run() {
        try {
                for(int loop_index = 0; loop_index < 4; loop_index++) {
                        System.out.println((Thread.
```

```
                        currentThread()).getName()+ " thread here...");
                        Thread.sleep(1000); }
            }
            catch (InterruptedException e) { }
            System.out.println((Thread.currentThread()).getName() + "
            ending."); }
    }
    class isAliveDemo
    {
      public static void main(String args[])
      {
            CustomThread thread1 = new CustomThread("first");
            CustomThread thread2 = new CustomThread("second");
            CustomThread thread3 = new CustomThread("third");
            CustomThread thread4 = new CustomThread("fourth");
            System.out.println("Thread1 is alive- "+thread1.isAlive());
            try {
                    thread1.join();
                    thread2.join();
                    thread3.join();
                    thread3.join(); }
            catch (InterruptedException e) { }
            System.out.println(thread1.isAlive()); }
    }
```

The output of this code is shown below.

Note the reported value of `isAlive()`, which is true when `thread1` is running and False after that thread has quit:

```
D:\Java8\chapter19>java isAliveDemo
first thread here...
third thread here...
Thread1 is alive- true
second thread here...
fourth thread here...
first thread here...
third thread here...
second thread here...
fourth thread here...
first thread here...
third thread here...
fourth thread here...
second thread here...
third thread here...
first thread here...
second thread here...
fourth thread here...
first ending.
third ending.
second ending.
fourth ending.
false
```

Setting Thread Priority and Stopping Threads

"Hmm," says the NP, "I'm getting the hang of threads, but there's a problem. I'm working on my new program, *SuperDuperMultiThreaded DataCrunch*, and some tasks are more important than others. For instance, I want to download images the user has asked before doing any spelling check for a document." "Well," you say, "you can give threads different *priorities*." The NP says, "Tell me more!"

Each thread in Java has a priority, which is between `Thread.MAX_PRIORITY` and `Thread.MIN_PRIORITY`. Here are the priorities that are currently defined:

- `MAX_PRIORITY`—The maximum priority that a thread can have

- `MIN_PRIORITY`—The minimum priority that a thread can have
- `NORM_PRIORITY`—The default priority assigned to a thread

The value of `Thread.MAX_PRIORITY` is 10, `Thread.MIN_PRIORITY` is 1, and `NORM_PRIORITY` is 5.

Setting priorities is a very tricky business because it's operating system dependent, and the actual results depend very much on what else is running at the same time. Nonetheless, you can set the relative priorities of threads. Here's an example in which we give four threads different priorities and just have them count for 5 seconds. Then, we end them and display their results to get an indication of their relative execution speeds. Here's the code (note that to stop the threads, we are using a flag that's checked in the `run()` method and is defined as `volatile`, which means its value can be changed by another thread):

```
class Counter implements Runnable {
  Thread thread;
  int counter = 0;
  volatile boolean goflag;
  public Counter(int p) {
        thread = new Thread(this);
        thread.setPriority(p); }
  public void start() {
        goflag = true;
        thread.start(); }
  public void run() {
        while (goflag)
        counter++; }
  public void end() { goflag = false; }
}
class PriorityDemo {
  public static void main(String args[]) {
        Counter thread1 = new Counter(Thread.NORM_PRIORITY + 2);
        Counter thread2 = new Counter(Thread.NORM_PRIORITY + 1);
        Counter thread3 = new Counter(Thread.NORM_PRIORITY - 1);
        Counter thread4 = new Counter(Thread.NORM_PRIORITY - 2);
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();
        try { Thread.sleep(5000); }
        catch (InterruptedException e) { }
        thread1.end();
        thread2.end();
        thread3.end();
        thread4.end();
        System.out.println("Thread 1 is counted: " + thread1.counter);
        System.out.println("Thread 2 is counted: " + thread2.counter);
        System.out.println("Thread 3 is counted: " + thread3.counter);
        System.out.println("Thread 4 is counted: " + thread4.counter); }
}
```

To stop the threads, we call the custom `end()` method, which sets a flag named `goflag` to False, which in turn stops the code in the `run()` method. In the early days of Java, you could stop a thread with a built-in `stop()` method, but that method is now deprecated because it's unsafe—it can leave operating system access monitors in an unstable state. Instead, to stop a thread, you now use the preceding method— that is, checking some flag that you can set from outside the thread.

In the output, note that the different number of counts from the threads with different priorities (your results may vary, of course, depending on your operating system and machine speed). The output of the code is shown in Figure 19.7, showing the threads in accordance with their priorities:

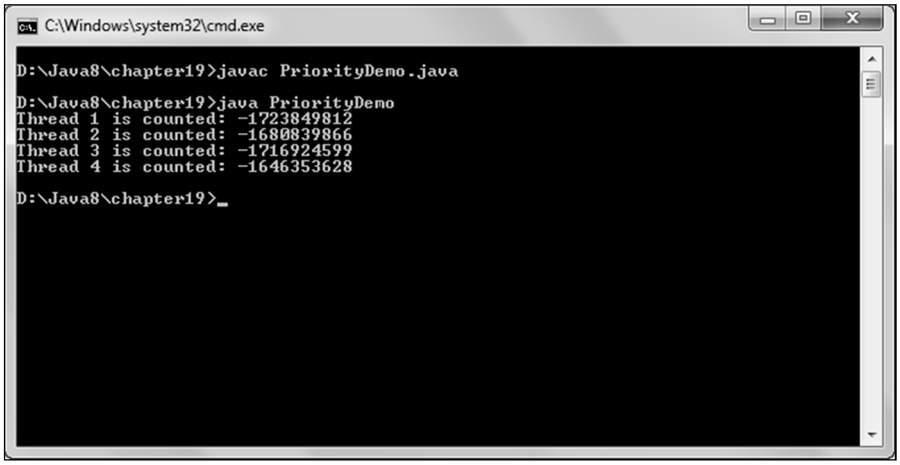


Figure 19.7: Counting the Threads with Different Priorities

Synchronizing

"Well," says the NP, "I guess I can't use threads anymore. I have one object that stores the data in my program, and I can't have many different threads working with that same object at the same time—what if one thread changes part of the data object and then another thread changes the same part of the data object before the first thread is finished? Chaos!" "Well," you smile, "you can prevent new threads from working with objects like that until the current thread is finished by synchronizing your threads." "Wow!" says the NP.

Excluding other threads from an operation until that operation has completed is called *synchronization*. Here's an example that shows why synchronization is useful. In this case, we have four new threads, all working with one shared data object. This object prints out a start message and an end message, and it completes a half-second task between the messages. Ideally, the data object should start, perform its task, and end before the next thread starts a new task but that's not the way it works here—each thread grabs the data object for itself. Here's the code:

```
class SynchronizeDemo
{
  public static void main(String args[])
  {
        Shared shared = new Shared();
        CustomThread thread1 = new CustomThread(shared, "one");
        CustomThread thread2 = new CustomThread(shared, "two");
        CustomThread thread3 = new CustomThread(shared, "three");
        CustomThread thread4 = new CustomThread(shared, "four");
        try {
                thread1.join();
                thread2.join();
                thread3.join();
                thread4.join(); }
        catch(InterruptedException e) { }
  }
}
class CustomThread extends Thread {
  Shared shared;
  public CustomThread(Shared shared, String string) {
        super(string);
        this.shared = shared;
        start(); }
  public void run() { shared.doWork(Thread.currentThread().getName()); }
}
class Shared {
```

```
    void doWork(String string) {
          System.out.println("Starting " + string);
          try { Thread.sleep((long) (Math.random() * 500)); }
          catch (InterruptedException e) {}
          System.out.println("Ending " + string); }
}
```

The output of this code is shown in Figure 19.8, showing the synchronization between the threads at various intervals:

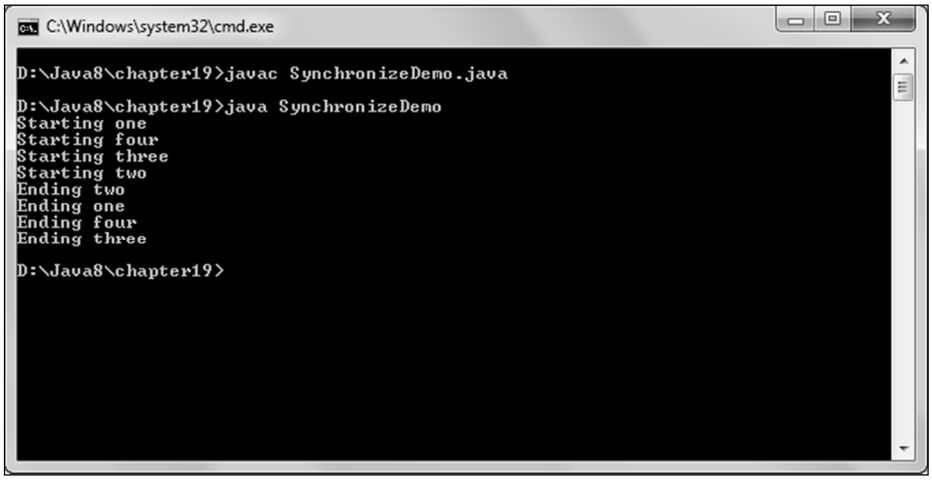


Figure 19.8: Synchronization between Threads

Note that each task overlaps the other.

This is a problem if you are working with actual data—what if you read some data from the data object, worked on it, and were about to rewrite it, but another thread has already done the same thing? When you rewrite the data, the changes from the other thread will be lost. As you can see, it's important to be able to block access to other threads to critical resources at times. Take a look at the next two solutions to see how this works.

Synchronizing Code Blocks

"OK," says the NP, "I know there can be a problem when multiple threads access the same object or resource, but how do I block the threads' access to new threads until the current one is done with that object or resource?" "There are two ways," you say. "OK," the NP says. "What's the first way?"

The two ways to synchronize thread code execution are synchronizing blocks of code and synchronizing methods. We'll take a look at synchronizing blocks of code in this solution and synchronizing methods in the next. To synchronize a block of code, you use the `synchronized()` method, indicating the object you want to restrict access to. Extending the example from the previous solution, the process looks like this:

```
class SynchronizeDemo1 {
  public static void main(String args[]) {
        Shared shared = new Shared();
        CustomThread thread1 = new CustomThread(shared, "two");
        CustomThread thread2 = new CustomThread(shared, "one");
        CustomThread thread3 = new CustomThread(shared, "five");
        CustomThread thread4 = new CustomThread(shared, "four");
        try {
                thread1.join();
                thread2.join();
                thread3.join();
                thread4.join(); }
        catch(InterruptedException e) { }
  }
}
class CustomThread extends Thread {
```

```
    Shared shared;
    public CustomThread(Shared shared, String string) {
          super(string);
          this.shared = shared;
          start(); }
    public void run() {
          synchronized(shared) {
                      shared.doWork(Thread.currentThread().getName()); }
    }
  }
  class Shared {
    void doWork(String string) {
          System.out.println("Stop " + string);
          try { Thread.sleep((long) (Math.random() * 500)); }
          catch (InterruptedException e) { }
  }
  }
```

The output of this code is shown in Figure 19.9, showing the synchronization between code blocks:

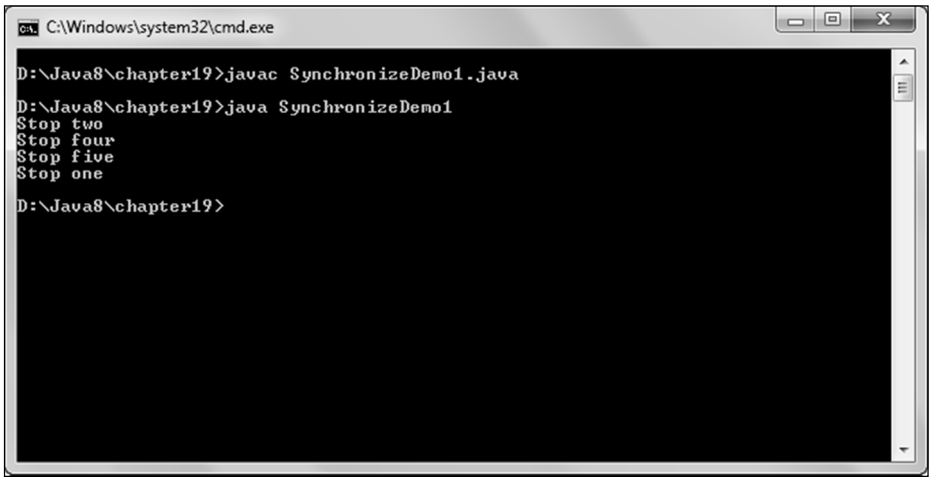


Figure 19.9: Synchronization between Code Blocks

Note that each task starts and ends before any other thread starts a new task.

Synchronizing Methods

Besides synchronizing code blocks, you can also block access to other threads by synchronizing methods. In this case, you use the `synchronized` keyword when defining a method. Here's how we modify the `doWork()` method in the `Shared` class and synchronize the threads:

```
  class SynchronizeDemo2 {
    public static void main(String args[]) {
          Shared shared = new Shared();
          CustomThread thread1 = new CustomThread(shared, "one");
          CustomThread thread2 = new CustomThread(shared, "two");
          CustomThread thread3 = new CustomThread(shared, "three");
          CustomThread thread4 = new CustomThread(shared, "four");
          try {
                  thread1.join();
                  thread2.join();
                  thread3.join();
                  thread4.join(); }
          catch(InterruptedException e) { }
    }
  }
  class CustomThread extends Thread {
```

```
    Shared shared;
    public CustomThread(Shared shared, String string) {
          super(string);
          this.shared = shared;
          start(); }
    public void run() { shared.doWork(Thread.currentThread().getName()); }
}
class Shared {
    synchronized void doWork(String string) {
          System.out.println("Starting " + string);
          try { Thread.sleep((long) (Math.random() * 500)); }
          catch (InterruptedException e) { }
          System.out.println("Ending " + string); }
}
```

Now, only one thread at a time can enter the `doWork()` method. The output of this code is shown in Figure 19.10:

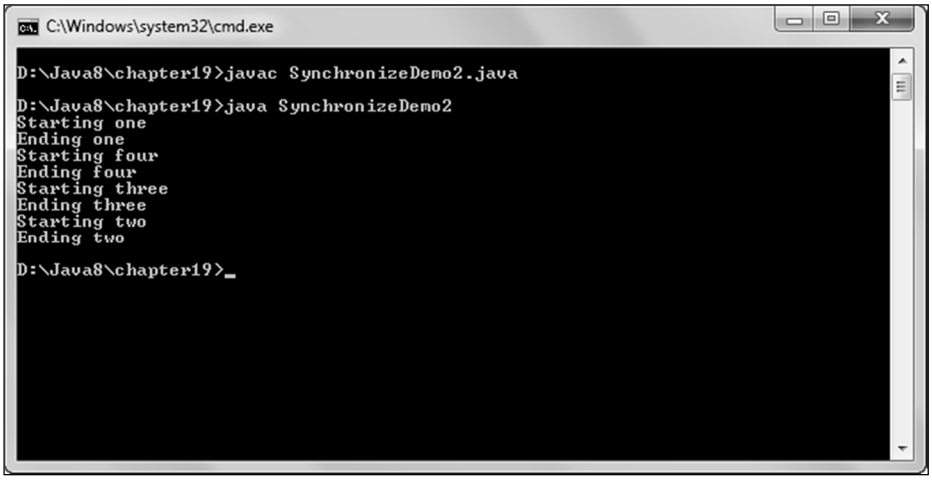


Figure 19.10: Synchronization between Methods

Let us look into another program that uses `synchronized()` method. In this program, in order to enter an object's monitor, just call a method that has been modified with the synchronized keyword. While a thread is inside a `synchronized()` method, all other threads that try to call it on the same instance have to wait. To relinquish control to the next waiting thread, the object returns from the `synchronized()` method:

```
class Callme {
    void call(String msg) {
          System.out.print("**" + msg);
          try { Thread.sleep(1000); }
          catch (InterruptedException e) { System.out.println("Stop..."); }
          System.out.println("****"); }
}
class Caller implements Runnable {
    String ms;
    Callme target;
    Thread h;
    public Caller(Callme targ, String s) {
          target = targ;
          ms = s;
          h = new Thread(this);
          h.start(); }
    public void run() { synchronized(target) { target.call(ms); } }
}
class Sync {
    public static void main(String args[]) {
          Callme target = new Callme();
```

```
            Caller c1 = new Caller(target, "Your");
            Caller c2 = new Caller(target, "JAVA");
            Caller c3 = new Caller(target, "Synchronized");
            try {
                    c1.h.join();
                    c2.h.join();
                    c3.h.join(); }
            catch(InterruptedException e) { System.out.println("Stop..."); }
      }
    }
```

The result of the code is shown below:

```
D:\Java8\chapter19>java Sync
**Your****
**JAVA****
**Synchronized****
```

Synchronizing Classes

Synchronization gives you the choice of adding the synchronized keyword to a method or wrapping a code section in a synchronized block. Several mechanisms for coordinating between different threads in an application have been added. Another useful category of classes in `java.util.concurrent` is the synchronizers. This set of classes coordinates and controls the flow of execution for one or more threads.

The `Semaphore`, `CyclicBarrier`, `CountdownLatch`, and `Exchanger` classes are all examples of synchronizers. These classes have methods that are called by threads, which may or may not block on the basis of the state and rules of the specific synchronizer being used.

The Semaphore Class

Prior to acquiring an item, a thread must obtain a permit from the semaphore, ensuring that an item will be available for use. Once the thread finishes with the item, it goes back to the pool, and a permit is given back to the semaphore, enabling another thread to acquire that item. The semaphore encapsulates the synchronization needed to restrict permit to the pool, separately from any synchronization required for maintaining the consistency of the pool itself.

Consider an example of a queue of people who came for their Aadhaar card registration. Each of them will be allotted a token number. The next day, they will be called according to their token number. A person who got a token number has occupied a certain position that is locked by him/her, and no other person shall be granted the same token number.

So, we can say that a semaphore restricts multiple numbers of users accessing the same token.

Let us discuss some of the terms related to synchronization. They are lock, permit, and block.

A lock is created so as to control the access of multiple threads over the same resource. It provides exclusive access to a shared resource, which means it allows only one thread to access an object at a time.

A permit is like a token to a lock that allows the thread to access a resource. A thread must have a permit from the semaphore before obtaining a resource. When the thread stops, it returns the permit to the semaphore, allowing another thread to access next.

A block indicates that a thread is waiting for the lock or permit to access the resource.

With those definitions in mind, let's look at the new `Semaphore` class, the class in Java that offers support for semaphores. You create the object of class with one of the following two constructors presented in Table 19.4:

Table 19.4: Constructors of the Semaphore class

Constructor	Does this
`Semaphore(int permits)`	It creates a semaphore with a given number of permits and non-fair fairness settings
`Semaphore(int permits, boolean fair)`	It creates a semaphore with a given number of permits and the given fairness settings

The methods of the Semaphore class and their functions are listed in Table 19.5:

Table 19.5: Methods of the Semaphore class

Method	Does this
`void acquire()`	It acquires a permit from this semaphore, blocking until one is available, or the thread is interrupted
`void acquire(int permits)`	It acquires the given number of permits from this semaphore, blocking until all are available, or the thread is interrupted
`void acquireUninterruptibly()`	It acquires a permit from this semaphore, blocking until one is available
`void acquireUninterruptibly(int permits)`	It acquires the given number of permits from this semaphore, blocking until all are available
`int availablePermits()`	It returns the current number of permits available in this semaphore
`int drainPermits()`	It acquires and returns all permits that are immediately available
`protectedCollection<thread> getQueuedThreads()`	It returns a collection containing threads that may be waiting to acquire
`int getQueueLength()`	It returns an estimate of the number of threads waiting to acquire
`boolean hasQueuedThreads()`	It queries whether any threads are waiting to acquire
`boolean isFair()`	It returns true if this semaphore has fairness set true
`protectedvoid reducePermits (int reduction)`	It shrinks the number of available permits by the indicated reduction
`void release()`	It releases a permit, returning it to the semaphore
`void release(int permits)`	It releases the given number of permits, returning them to the semaphore
`string toString()`	It returns a string identifying this semaphore as well as its state
`boolean tryAcquire()`	It acquires a permit from this semaphore, only if one is available at the time of invocation
`booloean tryAcquire(int permits)`	It acquires the given number of permits from this semaphore, only if all are available at the time of invocation
`boolean tryAcquire(int permits, long timeout, TimeUnit unit)`	It acquires the given number of permits from this semaphore, if all become available within the given waiting time, and the current thread has not been interrupted
`boolean tryAcquire(long timeout, TimeUnit unit)`	It acquires a permit from this semaphore, if one becomes available within the given waiting time, and the current thread has not been interrupted

The created semaphore has a "non-fair" setting. This means that First In First Out (FIFO) behavior is not guaranteed. To prevent starvation (a thread never getting a resource), it is better to create a fair semaphore. However, there are throughput advantages if you ignore fairness. To acquire a permit from a semaphore, the following two methods are invoked:

```
acquire()
acquire(int permits)
```

The `acquire()` method gets a single permit.

The `acquire(int permits)` method gets the requested number of permits. The methods block until a permit is available, or the waiting thread is interrupted. If the thread is interrupted, an `InterruptedException` is thrown.

The next two methods we are going to discuss, too, acquire permits, but their threads are dormant and uninterruptible until the required numbers of permits are available. If a thread is interrupted while waiting, it is regained after the resource becomes available:

```
acquireUninterruptibly()
acquireUninterruptibly(int permits)
```

The `tryAcquire()` method does not block. If the required number of permits is available, the methods return true. If the permits are not available, false is returned. All permits must be true to get available. The fairness setting is ignored with these untimed `tryAcquire()` methods:

```
tryAcquire()
tryAcquire(int permits)
```

The timed versions work similar to the untimed version, but wait for the specified `TimeUnit` before returning. If interrupted while waiting, the methods throw an `InterruptedException`:

```
tryAcquire(long timeout, TimeUnit unit)
tryAcquire(int permits, long timeout, TimeUnit unit)
```

The following line shows how to wait 40 seconds for one permit:

```
boolean acq = tryAcquire(40, TimeUnit.SECONDS)
```

Now, let's look at an example that uses the `Semaphore` class. The following program, `TestingSemaphore`, matches with an auction house service system. The service allows customers to see multiple items available for purchase and displays an average of the prices of these items. However, due to licensing limitations, only two threads can access the service at a time. To simulate the price sampling and averaging, the `TestingSemaphore` program waits 30 milliseconds and then returns a random number from 0 to 100. If the service is concurrently accessed after the limit of two threads is reached, a second pricing scheme is used. This latter scheme doesn't have the licensing restriction but returns a less-reliable price (in this example, $20). Here is the `TestingSemaphore` program:

```
import java.util.concurrent.*;
import java.util.*;
public class TestingSemaphore {
  private static final int COUNT_LOOP = 100;
  private static final int MAX_AVAILABLE = 2;
  private final static Semaphore sphore = new
  Semaphore(MAX_AVAILABLE, true);
  private static class Price {
        private static final Random rnd = new Random();
        public static int getPrice() {
                int prc = rnd.nextInt(100);
                try { Thread.sleep(30); }
                catch (InterruptedException ex) { ex.printStackTrace(); }
                return prc; }
        public static int getunfavPrice() { return 20; }
  }
  public static void main(String args[]) {
        for (int i=0; i< COUNT_LOOP; i++) {
                final int count = i;
                new Thread() {
                        public void run() {
                                int prc;
                                if (sphore.tryAcquire()) {
                                        try { prc = Price.getPrice(); }
                                        finally { sphore.release(); }
                                } else { prc = Price.getunfavPrice(); }
                                System.out.println(count + ": " + prc); }
                }.start();
        }
  }
}
```

Depending on the speed of your machine, you might need to modify the 50 millisecond delay for getting reasonable results. "Reasonable results" means that you get some random values generated and some $20 values. One sample runs as shown in Figure 19.11:

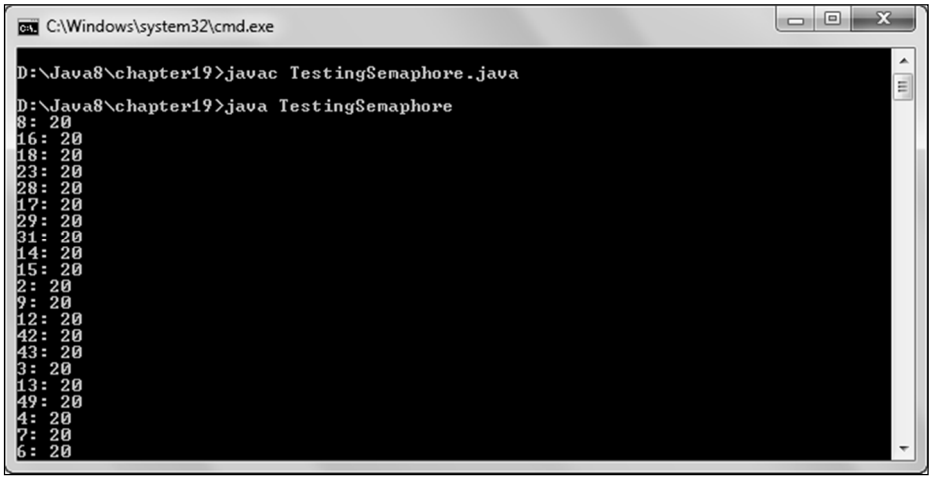


Figure 19.11 :Working of Semaphore

Your results might differ.

Notice how the results for pass 0 and 1 don't appear until after many other results. You then get more 20s before seeing more "good" results. Each call to the average pricing service takes time, therefore the interspersed results.

With the use of a semaphore, the program controls access to the limited resources. MUTEX-Mutual Exclusion semaphore or mutex is a special case of semaphore. Semaphores that allow an arbitrary resource count are called **counting semaphores**, while semaphores which are limited to the values 0 and 1 (or locked/unlocked, unavailable/available) are called **binary semaphores** (same functionality that mutexes have).

There are many features that are common in both mutexes and locks, except that mutexes have one additional feature that locks fail to have. It is the ability of the mutex to be released by a thread that is different from the one holding the permit.

The CyclicBarrier Class

The next synchronization pillar is the barrier. A barrier allows a common point (called a barrier point) for a set of threads to wait for each other before continuing their execution. The class that facilitates this functionality is called `CyclicBarrier`. This class allows one set of threads to pass through the barrier point till the time the next set of threads arrives without the requirement to create a new barrier.

An example used to define barriers is the splitting of a task into subtasks or segments such that each subtask or segment can be executed separately. For example, imagine a two-dimensional array or matrix, and you want the sum of all the values in the matrix. If you have a multiprocessor system, the splitting of the matrix will be faster. By dividing it into two halves, each processor sums up the values in its half. Then, when both sums are done, you add the sums together and return the combined total. That's exactly the use of a barrier. The barrier restricts the final addition till the addition of both halves is calculated.

The `CyclicBarrier` class allows you to create a barrier in one of the two ways. You can make a barrier with a number of threads, or with a number of threads and a barrier action. A barrier action is a `Runnable` task that executes when all the threads have joined together but before the threads are released to run on their own.

```
    CyclicBarrier(int parties)
    CyclicBarrier(int parties, Runnable barrierAction)
```

After each of the parties finishes its individual task, the party joins the barrier by calling its `await()` method. This blocks that party until all parties are present. If a barrier action is provided, it runs before releasing all the parties.

To demonstrate, the following code, App.java, performs a parallel summation of the elements of a triangular matrix:

```
import java.util.concurrent.*;
public class App {
  private static int matrices[][] = {
        {1},
        {2, 2},
        {3, 3, 3},
        {4, 4, 4, 4},
        {5, 5, 5, 5, 5}
  };
  private static int res[];
  private static class Summer extends Thread {
        int row;
        CyclicBarrier barrier;
        Summer(CyclicBarrier barrier, int row) {
                this.barrier = barrier;
                this.row = row; }
        public void run() {
                int cols = matrices[row].length;
                int sum = 0;
                for (int i=0; i<cols; i++) { sum += matrices[row][i]; }
                res[row] = sum;
                System.out.println("Results for row " + row + " are : " +
                sum);
                try { barrier.await(); }
                catch (InterruptedException ex) { ex.printStackTrace(); }
                catch (BrokenBarrierException ex) { ex.printStackTrace(); }
        }
  }
  public static void main(String args[]) {
        final int rows = matrices.length;
        res = new int[rows];
        Runnable merger = new Runnable() {
                public void run() {
                        int sum = 0;
                        for (int i=0; i<rows; i++) { sum += res[i]; }
                        System.out.println("Results are: " + sum); }
        };
        CyclicBarrier barrier = new CyclicBarrier(rows, merger);
        for (int i=0; i<rows; i++) { new Summer(barrier, i).start(); }
        System.out.println("Waiting..."); }
}
```

Compile and run the program. The result is shown in Figure 19.12:

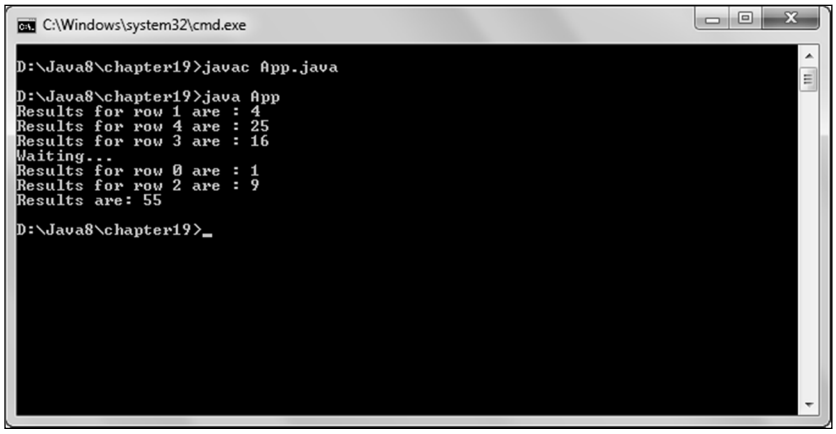


Figure 19.12: Printing Rows and Columns by Using the CyclicBarrier Class

Creating a new thread and scheduling it doesn't necessarily cause the system to pause the existing thread. Multiple threads might be created, all with the same priority. The system simply picks one thread to schedule at that priority level. If you wish to reuse a cyclic barrier, you can call the reset method of `CyclicBarrier` between usages.

The CountDownLatch Class

The next synchronization aid is called a latch. Latching variables specify conditions that never change after being set. This provides a way to start several threads and have them wait until a signal is received from a coordinating thread. Latching works well with tasks where you want no process to run until everything needed is initialized. Latching also works well for multiplayer games, where you don't want any player to start until all players have joined.

The `CountDownLatch` class encapsulates a latch that is initialized with a given count. The constructor takes the count as its one argument. The count works in a way similar to the threads argument to the `CyclicBarrier` constructor. It indicates how many times a `countDown()` method must be called, one for each party involved. After the full count is reached, any threads waiting, because of an `await()` method, are released.

The following program, `TestingLatch`, demonstrates the use of a `CountDownLatch`. The program creates a set of threads but doesn't let any thread start until all the threads are created:

```
import java.util.concurrent.*;
public class TestingLatch {
  private static final int COUNT = 5;
  private static class Work implements Runnable {
        CountDownLatch Latch_start;
        CountDownLatch Latch_stop;
        String name;
        Work(CountDownLatch Latch_start, CountDownLatch Latch_stop,
        String name) {
                this.Latch_start = Latch_start;
                this.Latch_stop = Latch_stop;
                this.name = name; }
        public void run() {
                try { Latch_start.await(); }
                catch (InterruptedException ex) { ex.printStackTrace(); }
                System.out.println("Running: " + name);
                Latch_stop.countDown(); }
  }
  public static void main(String args[]) {
        CountDownLatch Signal_start = new CountDownLatch(1);
        CountDownLatch Signal_stop  = new CountDownLatch(COUNT);
        for (int i=0; i<COUNT; i++) {
                new Thread(new
Work(Signal_start, Signal_stop,Integer.toString(i))).start(); }
        System.out.println("Go");
        Signal_start.countDown();
        try { Signal_stop.await(); }
        catch (InterruptedException ex)
        { ex.printStackTrace(); }
        System.out.println("Done"); }
}
```

The main thread waits on a second latch to make sure that other threads are finished. This example could have simply used a join operation on the created threads; however, imagine if you needed to synchronize threads at different phases through their lifetime, instead of just at the end. The result is shown in Figure 19.13:

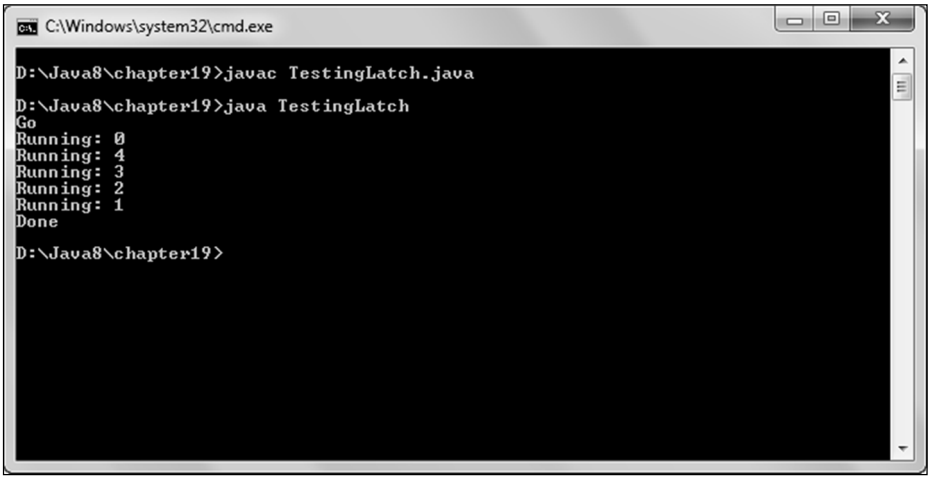


Figure 19.13: Using the `CountDownLatch` Class

The Exchanger Utility

The Exchanger is the last of the synchronization utilities covered in this chapter. It offers a simple way of communicating between threads, that is, by passing a specific object between two threads. That's why there's `<V>` after the class name. Instead of using the Piped streams for stream-based, inter-thread communication (where one side writes and the other reads), the Exchanger code relies on a single `exchange()` method for the transfer of one-off data between threads. The Exchanger constructor takes no arguments here, other than identifying the type of object to exchange:

```
Exchanger<List<Integer>> exchanger = new Exchanger<List<Integer>>();
```

To exchange objects, you call the `exchange()` method. The method transfers objects in both directions, not just one:

```
V exchange(V x)
```

When one of the threads is consuming a resource and the other one is producing, an exchanger is used. When the buffer used by the producer is full, the producer waits for the consumer. When the buffer consumed by the consumer is empty, it waits for the producer. After both waits occur, the two threads swap buffers.

Here's an example that uses the Exchanger. The `FillLoop` class is the producer type. The `EmptyLoop` class is the consumer. When the producer's data structure is full, it tries to exchange with the consumer. When the consumer's data structure is empty, it tries to exchange data structures with the producer. When both the producer and consumer are waiting, the exchange happens.

```
import java.util.*;
import java.util.concurrent.*;
public class ExchangerDemo {
  private static final int FULL = 10;
  private static final int COUNT = FULL * 20;
  private static final Random rnd = new Random();
  private static volatile int sum = 0;
  private static Exchanger<List<Integer>> exchanger =
  new Exchanger<List<Integer>>();
  private static List<Integer> EmptyBuffer;
  private static List<Integer> FullBuffer;
  private static CountDownLatch Latch_stop = new CountDownLatch(2);
  private static class FillLoop implements Runnable {
        public void run() { List<Integer> currentBuf = EmptyBuffer;
                try {
                        for (int i = 0; i < COUNT; i++) {
                                if (currentBuf == null)
                                break;
                                Integer item = rnd.nextInt(100);
```

```
                                    System.out.println("Added: " + item);
                                    currentBuf.add(item);
                                    if (currentBuf.size() == FULL)
                                    currentBuf = exchanger.exchange(currentBuf);
                            }
                    }
                    catch (InterruptedException ex) {
                            System.out.println("Bad exchange on filling side");}
                    Latch_stop.countDown(); }
      }
      private static class EmptyLoop implements Runnable {
            public void run() {
                    List<Integer> currentBuf = FullBuffer; try {
                            for (int i = 0; i < COUNT; i++) {
                                    if (currentBuf == null) break;
                                    Integer item = currentBuf.remove(0);
                                    System.out.println("Got: " + item);
                                    sum += item.intValue();
                                    if (currentBuf.isEmpty()) {
                                            currentBuf =
                                            exchanger.exchange(currentBuf); }
                            }
                    }
                    catch (InterruptedException ex) {
                            System.out.println("Bad exchange on emptying side");
                    }
                    Latch_stop.countDown(); }
      }
      public static void main(String args[]) {
            EmptyBuffer = new ArrayList<Integer>();
            FullBuffer = new ArrayList<Integer>(FULL);
            for (int i=0; i<FULL; i++) { FullBuffer.add(rnd.nextInt(100)); }
            new Thread(new FillLoop()).start();
            new Thread(new EmptyLoop()).start();
            try { Latch_stop.await(); }
            catch (InterruptedException ex) { ex.printStackTrace(); }
            System.out.println("Sum of all items is.... " + sum); }
   }
```

The filling and emptying loops run a fixed number of times before stopping. They use a `CountDownLatch` to tell the main thread when they are done. The output includes sections of "Added" and "Got" values that should match, though the actual numbers might differ in your runs. The result is shown in Figure 19.14:



Figure 19.14: Using the Exchanger Class

Communicating between Threads

"No!" says the NP "I have got a threading problem. Part of my program is writing data and part of it is reading data—but sometimes, the reading part gets ahead of the writing part!" "That's a classic producer/consumer problem," you say, "and you can fix it with the `wait()` and `notify()` methods." The NP smiles and says, "You can?"

Often, threads will need to coordinate between themselves, especially when the output of one thread is used by another thread. One way of coordinating threads is to use the `wait()`, `notify()`, and `notifyAll()` methods:

- `wait()`—It makes a thread sleep until `notify()` or `notifyAll()` method is called.
- `notify()`—It starts the first thread that called `wait()` on the same object.
- `notifyAll()`—It starts all the threads that called `wait()` on the same object.

The usual process is for the reader thread to call `wait()` method and the writer thread to call `notify()` method when the data the reader wants to read is ready. Here's an example. In this case, a writer thread will call an object's `doWork()` method, which is a time-consuming method, to write some data, and a reader thread will call the same object's `getResult()` method to read the results. Clearly, we want the reader thread to have to wait until `doWork()` method is done. Therefore, all we have to do is call `wait()` in `getResult()` method to make the reader thread wait and call `notify()` method in `doWork()` when the writer thread is done, and the data is ready to be read. Here's what the code looks like:

```
class Shared {
  int data = 0;
  synchronized void doWork() {
        try { Thread.sleep(1000); }
        catch(InterruptedException e) { }
        data = 1;
        notify(); }
  synchronized int getResult() {
        try { wait(); }
        catch(InterruptedException e) { }
        return data; }
}
class CustomThread1 extends Thread {
  Shared shared;
  public CustomThread1 (Shared shared, String string) {
        super(string);
        this.shared = shared;
        start(); }
  public void run() {
        System.out.println("The result is " + shared.getResult()); }
}
class CustomThread2 extends Thread {
  Shared shared;
  public CustomThread2 (Shared shared, String string) {
        super(string);
        this.shared = shared;
        start(); }
  public void run() { shared.doWork(); }
}
class Waiting {
  public static void main(String args[]) {
        Shared shared = new Shared();
        CustomThread1 thread1 = new CustomThread1(shared, "one");
        CustomThread2 thread2 = new CustomThread2(shared, "two"); }
}
```

Here's the output of this code. Note that the reader thread did wait until the writer thread was through:

```
C:\>java Waiting
The result is 1
```

Suspending and Resuming Threads

In the early days of Java, threads supported both a `suspend()` method and a `resume()` method, which you could use to temporarily halt and start a thread again. Unfortunately, like the `stop()` method, both `suspend()` and `resume()` methods have been deprecated. In this case, these methods are deprecated because they are deadlock-prone and can block access to other threads. However, you can create your own `suspend()` and `resume` methods, and we'll show how to do that here.

Note that when sub-classing the `Thread` class, you can't call your new "suspend" and "resume" methods `suspend()` and `resume()`, because Java will complain that you are trying to override deprecated methods. Therefore, in this example, we'll call these methods `newSuspend()` and `newResume()` (this isn't a problem when working with an object that implements the `Runnable` interface instead). As is the case when you implement a new `stop()` method (see the solution "Setting Thread Priority and Stopping Threads" earlier in this chapter), these methods work by setting a flag that's checked in the `run()` method. When you set that flag to true in the `newSuspend()` method, you use the `wait()` method in the `run` method to suspend the thread, and you wake the thread up again with the `notify()` method in the `newResume()` method. Here's how this looks in code:

```
class CustomThread extends Thread {
  volatile boolean goFlag = true;
  CustomThread(String name) {
        super(name);
        start(); }
  public void run() {
        try {
                for(int loop_index = 0; loop_index <=3; loop_index++) {
                        System.out.println(Thread.currentThread().getName()
                        + "here...");
                        Thread.sleep(500);
                        synchronized(this) {
                                while(!goFlag) { wait(); }
                        }
                }
        }
        catch (InterruptedException e) { }
  }
  public void newSuspend() { goFlag = false; }
  synchronized public void newResume() {
        goFlag = true;
        notify(); } }
class SuspendDemo {
  public static void main(String args[]) {
        CustomThread thread1 = new CustomThread("one");
        CustomThread thread2 = new CustomThread("two");
                try {
                        Thread.sleep(1000);
                        System.out.println("Suspending thread one...");
                        thread1.newSuspend();
                        Thread.sleep(1000);
                        System.out.println("Resuming thread one...");
                        thread1.newResume(); }
                catch (InterruptedException e) { }
                try {
                        thread1.join();
                        thread2.join(); }
                catch (InterruptedException e) { }
        }
  }
```

The output of this code is shown in Figure 19.15:

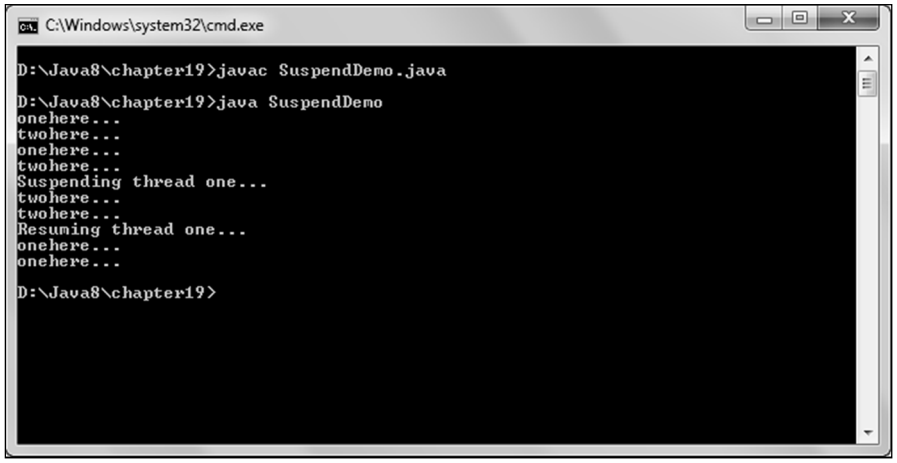


Figure 19.15: Suspending Threads

Note that threads one and two run until the code suspends thread one; then, when the code starts thread one again, they alternate again.

Even though the methods `suspend()` and `resume()` seem to be having a better approach, they should not be used. The function `suspend()` was deprecated by Java 2 because it could cause serious system failures. The `resume()` method was deprecated by Java 2 too. This was because it cannot be used without the `suspend()` method. The best way would be that we design a `run()` method that would periodically check whether the thread should suspend or resume. This can be accomplished using a `counter` variable. As long as this counter is set to running, the `run()` method must continue to execute. If this variable is set to suspend, the thread must pause. If it is set to stop, the thread must terminate:

```
class NewThread implements Runnable
{
   String SR;
   Thread h;
   boolean counter;
   NewThread(String threadname) {
          SR = threadname;
          h = new Thread(this, SR);
          System.out.println("Begin thread: " + h);
          counter = false;
          h.start(); }
   public void run() {
          try {
                 for(int i = 30; i > 15; i--) {
                        System.out.println(SR + ": " + i);
                        Thread.sleep(200);
                        synchronized(this) {
                               while(counter) { wait(); }
                        }
                 }
          }
          catch (InterruptedException e)
          {
                 System.out.println(SR + " Suspended..."); }
                 System.out.println(SR + " Ending..."); }
                 void mysuspend() { counter = true; }
                 synchronized void myresume()
                 {
                        counter = false;
                        notify();
```

```
                }
    }
    class SuspendResume {
      public static void main(String args[]) {
            NewThread ob1 = new NewThread("One");
            NewThread ob2 = new NewThread("Two");
            try {
                    Thread.sleep(1000);
                    ob1.mysuspend();
                    System.out.println("Suspending One...");
                    Thread.sleep(1000);
                    ob1.myresume();
                    System.out.println("Resuming One...");
                    ob2.mysuspend();
                    System.out.println("Suspending Two...");
                    Thread.sleep(1000);
                    ob2.myresume();
                    System.out.println("Resuming Two..."); }
            catch (InterruptedException e) {
                    System.out.println("Main thread Suspended..."); }
            try {
                    System.out.println("Waiting for threads to finish.");
                    ob1.h.join();
                    ob2.h.join(); }
            catch (InterruptedException e) {
                    System.out.println("Main thread Suspended"); }
            System.out.println("Main thread exiting..."); }
    }
```

The result is shown in Figure 19.16:

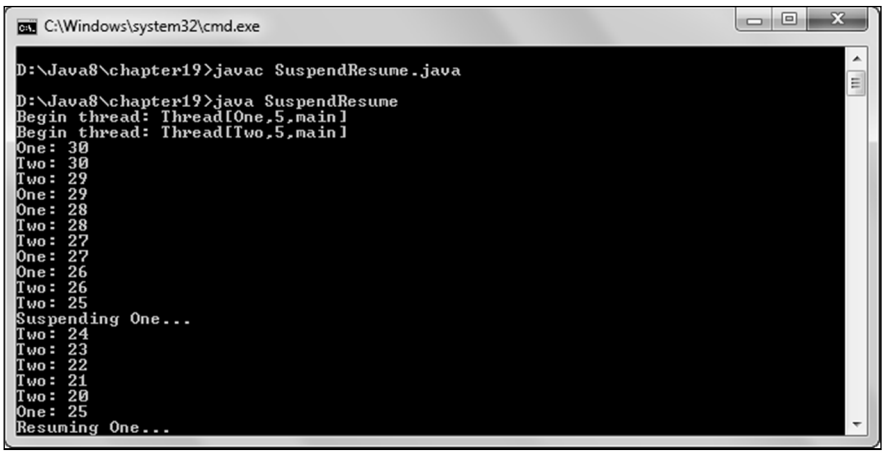


Figure 19.16: Resuming Threads

Creating Graphics Animation with Threads

"Hey," says the NP, "I just thought of something—using threads is ideal when you want to support graphics animation, isn't it?" "Sure is," you say. "Great," says the NP, "you write the code, and I'll watch."

Here's an example showing how graphics animation by using threads can work. In this applet, we load four images into an image array and then we cycle through them in the `paint()` method, making Java call that method repeatedly by calling `repaint()` in a new thread.

There's one interesting point here—although the `Thread` class's `stop()` method has been deprecated, Web browsers and the Oracle appletviewer still call this method when the page containing the applet is no longer the current page. To stop the applet's animation thread so that it doesn't keep taking up system resources until the

browser decides to unload it, you can place code in the `stop()` method yourself —but that means you can't base your code on the `Thread` class, because `stop()` has been deprecated there. Therefore, we'll write an example by using the `Runnable` interface, which doesn't have a deprecated `stop()` method. Here's the code:

```
import java.awt.*;
import java.applet.Applet;
/*<APPLET
  CODE = Whirll.class
  WIDTH = 300
  HEIGHT = 300>
</APPLET>*/
public class Whirll extends Applet implements Runnable {
  Image whirlImages[] = new Image[4];
  Image nowImage;
  int whirlIndex = 0;
  Thread whirlThread;
  boolean animateFlag = true;
  public void init() {
        whirlImages[0] = getImage(getCodeBase(), "whirl1.gif");
        whirlImages[1] = getImage(getCodeBase(), "whirl2.gif");
        whirlImages[2] = getImage(getCodeBase(), "whirl3.gif");
        whirlImages[3] = getImage(getCodeBase(), "whirl4.gif"); }
  public void start() {
        whirlThread = new Thread(this);
        whirlThread.start(); }
  public void stop() { animateFlag = false; }
  public void run() {
        while(animateFlag) {
                nowImage = whirlImages[whirlIndex++];
                if(whirlIndex > 3)whirlIndex = 0;
                repaint();
                try {Thread.sleep(200);}
                catch(InterruptedException e) { }
        }
  }
  public void paint (Graphics g) {
        if(nowImage != null)
        g.drawImage(nowImage, 10, 10, this); }
}
```

You can see the result of this code in Figure 19.17:

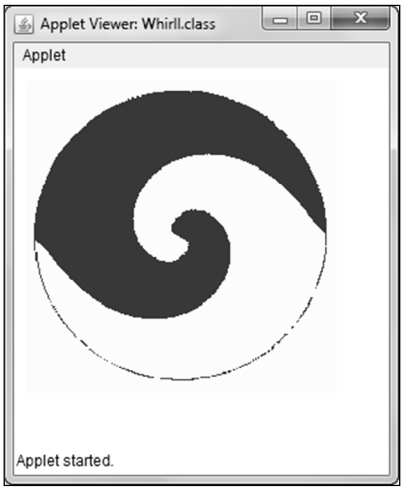


Figure 19.17: Displaying Graphics Animation

This applet makes the image you see in that figure. It appears to whirl around.

We'll create another example that would create a ball that travels across a component—starting from the left side of the component and moving toward the right:

```
import java.applet.*;
import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;
/*<APPLET
  CODE = MoveBall.class
  WIDTH = 300
  HEIGHT = 300>
</APPLET>*/
public class MoveBall extends JApplet {

  private final int WIDTH = 350;
  private final int HEIGHT = 300;
  private PaintSurface canvas;
  public void init() {
        this.setSize(WIDTH, HEIGHT);
        canvas = new PaintSurface();
        this.add(canvas, BorderLayout.CENTER);
        Thread t = new AnimationThread(this);
        t.start(); }
}
class AnimationThread extends Thread {
  JApplet c;
  public AnimationThread(JApplet c) { this.c = c; }
  public void run() {
        while (true) {
                c.repaint();
                try {Thread.sleep(30);}
                catch (InterruptedException ex) { }
        }
  }
}
class PaintSurface extends JComponent {
  int x_pos = 20;
  int y_pos = 50;
  int d = 20;
  public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_ON);
        x_pos += 4;
        Shape ball = new Ellipse2D.Float(
        x_pos, y_pos, d, d);
        g2.setColor(Color.YELLOW);
        g2.fill(ball); }
}
```

You can see the result of this code in Figure 19.18, where the applet moves the ball across—from left to right:

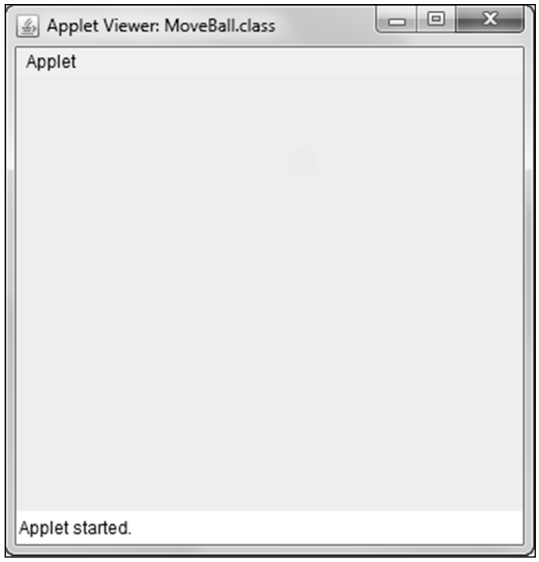


Figure 19.18: Ball Moving across the Applet

However, there's one problem in both these programs. Here, we have implemented these examples as the JApplet class, and each time the image is redrawn, the background behind the image is also redrawn, so the animation flickers a lot. Moreover, everything is drawn straight to the screen. To fix this problem, see the next solution.

Eliminating Flicker in Graphics Animation Created Using Threads

"Hmm," says the NP. "I can create graphics animation by using threads now, but it sure flickers a lot." "No problem," you say. "Just override the `update()` method." The NP asks, "How's that?"

In AWT programming, when you redraw an image, its background is redrawn first, which can make for a lot of flickering in graphics animation. The way to fix this is to override the `update()` method, which does the redrawing. When you override `update()`, the image's background is not redrawn. In fact, you can do more—you can make the `paint()` method only redraw the area covered by the image you want to redraw by using the `Graphics` class's `clipRect()` method, like this:

```
import java.awt.*;
import java.applet.Applet;
/*<APPLET
  CODE = Whirl2.class
  WIDTH = 300
  HEIGHT = 300 >
</APPLET>*/
public class Whirl2 extends Applet implements Runnable {
  Image whirlImages[] = new Image[4];
  Image nowImage;
  int whirlIndex = 0;
  Thread whirlThread;
  boolean animateFlag = true;
  public void init() {
        whirlImages[0] = getImage(getCodeBase(), "whirl1.gif");
        whirlImages[1] = getImage(getCodeBase(), "whirl2.gif");
        whirlImages[2] = getImage(getCodeBase(), "whirl3.gif");
        whirlImages[3] = getImage(getCodeBase(), "whirl4.gif");
  }
  public void start()
```

```
    {
          whirlThread = new Thread(this);
          whirlThread.start();
    }
    public void stop() { animateFlag = false; }
    public void run()
    {
          while(animateFlag)
          {
                nowImage = whirlImages[whirlIndex++];
                if(whirlIndex > 3)whirlIndex = 0;
                repaint();
                try { Thread.sleep(200); }
                catch(InterruptedException e) { }
          }
    }
    public void paint (Graphics g)
    {
          if(nowImage != null) g.drawImage(nowImage, 10, 10, this);
    }
    public void update(Graphics g) {
          g.clipRect(100, 100, 280, 280);
          paint(g); }
}
```

Now, the image shown in Figure 19.19 will rotate practically flicker-free:

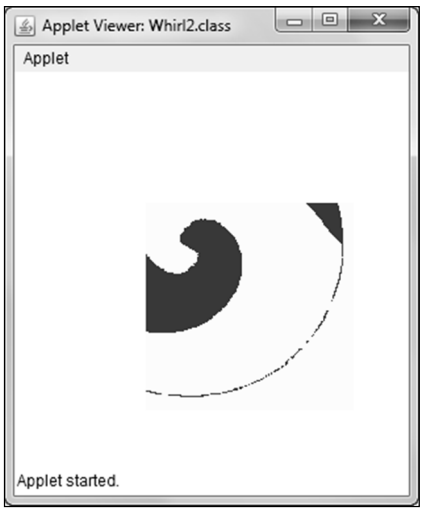


Figure 19.19: Rotation of the Image

Suspending and Resuming Graphics Animation

"What if I want to suspend and resume graphics animation? Can I do that?" the NP asks. "Sure," you say, "no problem. Just use the standard suspend and resume techniques." "Great," says the NP, "show me how."

Here's an example that adds a button to suspend and resume the animation developed over the previous two solutions. This puts to work the thread suspend and resume techniques (that is, using a boolean flag and the `wait()` and `notify()` methods) discussed in the solution "Suspending and Resuming Threads," earlier in this chapter:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
/*<APPLET
```

```
    CODE = Whirl3.class
    WIDTH = 300
    HEIGHT = 300 >
</APPLET>*/
public class Whirl3 extends Applet implements ActionListener, Runnable {
    Image whirlImages[] = new Image[4];
    Image nowImage;
    int whirlIndex = 0;
    Thread whirlThread;
    Button suspendButton, resumeButton;
    boolean animateFlag = true;
    boolean goFlag = true;
    public void init()
    {
          whirlImages[0] = getImage(getCodeBase(), "whirl1.gif");
          whirlImages[1] = getImage(getCodeBase(), "whirl2.gif");
          whirlImages[2] = getImage(getCodeBase(), "whirl3.gif");
          whirlImages[3] = getImage(getCodeBase(), "whirl4.gif");
          suspendButton = new Button("Suspend");
          add(suspendButton);
          suspendButton.addActionListener(this);
          resumeButton = new Button("Resume");
          add(resumeButton);
          resumeButton.addActionListener(this);
    }
    public synchronized void actionPerformed(ActionEvent e)
    {
          if(e.getSource() == suspendButton){goFlag = false;}
          if(e.getSource() == resumeButton) {
                goFlag = true;
                notify();
          }
    }
    public void start()
    {
          whirlThread = new Thread(this);
          whirlThread.start(); }
    public void stop() { animateFlag = false; }
    public void run() {
          while(animateFlag) {
                nowImage = whirlImages[whirlIndex++];
                if(whirlIndex > 3)whirlIndex = 0;
                repaint();
                try {
                      Thread.sleep(200);
                      synchronized(this) {
                            while(!goFlag)
                            wait(); }
                }
                catch(InterruptedException e) { }
          }
    }
    public void paint (Graphics g)
    {
          if(nowImage != null) g.drawImage(nowImage, 10, 10, this);
    }
    public void update(Graphics g)
    {
          g.clipRect(150,100,200,280);
          paint(g);
    }
}
```

The result of this code is shown in Figure 19.20:

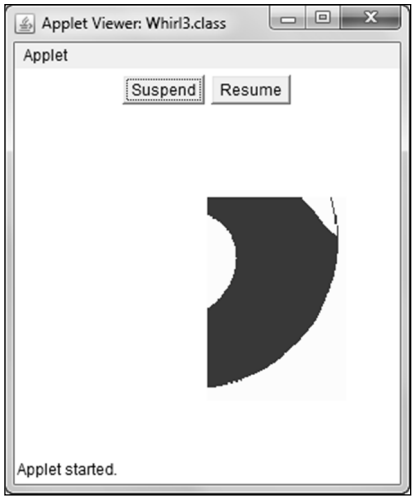


Figure 19.20: Displaying Graphics Animation with the Suspend and Resume Buttons

Using Double Buffering

Double buffering is the process of creating images off-screen and flashing them on screen as required, which means you don't have to create the image while the user watches. You can create a graphics buffer by creating an `Image` object and then using that object's `getGraphics()` method to get a `Graphics` object for the image that you can use to draw in. Here's an example showing how double buffering works. This example draws a succession of boxes off-screen and then displays them, thus creating an animation:

```
import java.awt.*;
import java.applet.Applet;
/*<APPLET
  CODE=DBuffer.class
  WIDTH=200
  HEIGHT=200>
</APPLET>*/
public class DBuffer extends Applet implements Runnable
{
  Image image1;
  Thread thread1;
  Graphics graphics;
  int loop_index = 0;
  boolean goFlag = true;
  public void init()
  {
        image1 = createImage(100, 100);
        graphics = image1.getGraphics();
  }
  public void start() {
        thread1 = new Thread(this);
        thread1.start();
  }
  public void stop() { goFlag = false; }
  public void run()
  {
        while(goFlag)
        {
                repaint();
                try { Thread.sleep(100); }
```

```
                    catch(InterruptedException e) { }
            }
      }
      public void paint (Graphics g) {
            loop_index += 5;
            if(loop_index >= 100) loop_index = 5;
            graphics.setColor(new Color(255, 255, 255));
            graphics.fillRect(0, 0, 50, 50);
            graphics.setColor(new Color(0, 255, 255));
            graphics.drawRect(0, 0, loop_index, loop_index);
            g.drawImage(image1, 10, 10, this); }
   }
```

The result of the code is shown in Figure 19.21:

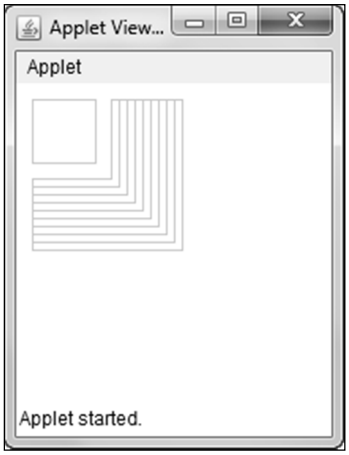


Figure 19.21: An Applet Showing a Succession of Boxes

Let us take a look at another code that will use a technique called double buffering, which will produce smooth, flicker-free animation. This technique first draws an off-screen image, and when everything is done, the off-screen image is drawn on the screen, thus reducing nasty flickering:

```
   import java.applet.*;
   import java.awt.event.*;
   import java.awt.*;
   /* <APPLET
      CODE = DoubleBuffering.class
      WIDTH = 300
      HEIGHT = 300>
   </APPLET>*/
   public class DoubleBuffering extends Applet implements MouseMotionListener {
      Graphics bufferGraphics;
      Image offscreen;
      Dimension dim;
      int curX, curY;
      public void init() {
            dim = getSize();
            addMouseMotionListener(this);
            setBackground(Color.white);
            offscreen = createImage(dim.width,dim.height);
            bufferGraphics = offscreen.getGraphics(); }
      public void paint(Graphics g) {
            bufferGraphics.clearRect(0,0,dim.width,dim.width);
            bufferGraphics.setColor(Color.gray);
            bufferGraphics.drawString("Double-buffered for you",2,10);
            bufferGraphics.fillRect(curX,curY,20,20);
            g.drawImage(offscreen,0,0,this); }
      public void update(Graphics g) { paint(g); }
```

```
    public void mouseMoved(MouseEvent evt) {
          curX = evt.getX();
          curY = evt.getY();
          repaint(); }
    public void mouseDragged(MouseEvent evt) { }
}
```

Update is called automatically when `repaint()` is called. The `update()` method is always required for good double buffering. By this, the applet immediately starts to repaint rather than wiping the previous drawings off first. Flickering is caused by the wiping-off action. The result of the code is shown in Figure 19.22:

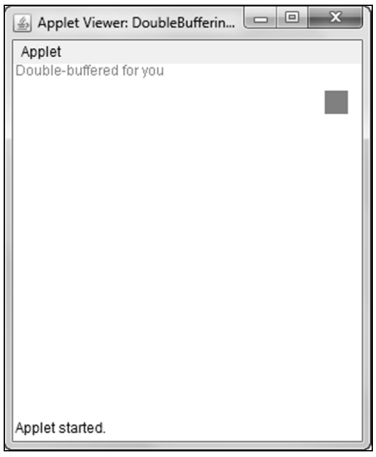


Figure 19.22: Applet Showing Flicker-Free Drawing

Simplifying Producer-Consumer with the **Queue** Interface

Until Java 5, we were rolling our own Queues. Now the util package comes with a new interface, `java.util.Queue`, with various implementations (including a retrofitted `LinkedList`). There are various queues, including Priority Queues, Blocking Queues, Delay Queues, and Synchronous Queues. We are here dealing with simplification of producer-consumer with the `java.util.Queue` interface.

One of the fundamental data structures in Computer Science is the queue. You'll recall that a queue is a data structure in which elements are removed in the same order in which they were added. Unfortunately, First in First Out (FIFO) data structure was ignored in the earlier versions of Java. After Java 1.5, queue support has become innate for the first time.

Queues are typically restricted to a given size.

For enqueuing and dequequing via the offer or add methods and the remove or poll methods respectively, if the queue fails to provide enqueuing or dequeuing operation, you need not wait for the execution of the program. The `java.util.concurrent.BlockingQueue` interface adds the `put()` and `take()` methods to the collection. `BlockingQueue` executions are designed to be used for producer-consumer queues. The relationship of producer/consumer can be understood as the producer writes to a queue (more specifically a `BlockingQueue`), just to suppose. There are a number of consumers reading from that queue, and you want all that to happen in a sequential manner. It implies that each consumer needs to wait for the previous consumer before it is allowed to grab an item from the queue. To create such a setup using code, you must spawn one producer thread that would simply write to a queue and also spawn several consumer threads that read from the same thread. It is to be taken into consideration that the threads will block one another until the current thread will grab the item from the queue.

The code here shows the class Producer writing to a `BlockingQueue`. Notice that objects in the `run()` method (which you are obligated to implement, since you extended Thread) are put onto the `BlockingQueue` after

waiting for a random amount of time (ranging from 100 – 500 milliseconds). The object put onto the queue is simply a `String` object consisting of the timestamp when the message was produced.

The actual work of enqueuing the object is done by the statement:

```
blockingQueue.put("Enqueued at: " + time)
```

The `put()` method throws an `InterruptedException`. Accordingly, the `put()` operation catches the operation thrown, which is surrounded by a `try` and a `catch` block. The messages that are thrown from the producer belong to the `Consumer` object, which extends from the `Thread` object and, therefore, must implement the `run()` method.

The `Consumer` class is very similar in design to the `Producer` class. The `Consumer` class makes use of the `take()` method to dequeue messages from the queue rather than putting messages on the `BlockingQueue`. As already stated, this occurs because the thread waits until something is actually on the queue. If, onto the queue, queuing of objects is stopped by the Producer thread, the Consumer thread will have to wait till there is an item available in the queue. The following code illustrates the `TestBlockingQueue` class that spawns four consumer threads, which try and grab objects from the `BlockingQueue`:

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.Date;
import java.util.Random;
import java.io.PrintStream;
class Producer extends Thread
{
        private BlockingQueue<String> blockQ;
        private PrintStream prnStr;
        public Producer(BlockingQueue<String> blockQ, PrintStream prnStr)
        {
                this.prnStr = prnStr;
                this.blockQ = blockQ;
        }
        public void run()
        {
                try {
                        while(true)
                        {
                                String time = String.format("%tc",new Date());
                                prnStr.println("Putting item on BlockingQueue
                                at time: " + time);
                                blockQ.put("Enqueued at: " + time);
                                Random generate = new Random();
                                int randomSlp = generate.nextInt(5) + 1;
                        Thread.sleep(randomSlp * 100);
                        }
                }
                catch (InterruptedException e) { e.printStackTrace(); }
        }
}
class Consumer extends Thread
{
  private BlockingQueue blockQ;
  private String consumName;
  private PrintStream prnStr;
  public Consumer(String consumName, BlockingQueue blockQ,
  PrintStream prnStr)
  {
        consumName = this.consumName;
        blockQ = this.blockQ; prnStr = this.prnStr; }
        public void run()
        {
        while (true) {
                try {
                prnStr.println(this.consumName + blockQ.take());
                }
```

```
                    catch (InterruptedException ie) { ie.printStackTrace(); }
                    }
            }
    }
    public class TestBlockingQueue
    {
       public static void main(String args[])
       {
              BlockingQueue<String> blkQue = new LinkedBlockingQueue<String>();
              Producer prod = new Producer(blkQue, System.out);
              Consumer consP = new Consumer("ConsumerP", blkQue, System.out);
              Consumer consQ = new Consumer("ConsumerQ", blkQue, System.out);
              Consumer consR = new Consumer("ConsumerR", blkQue, System.out);
              Consumer consS = new Consumer("ConsumerS", blkQue, System.out);
              prod.start();
              consP.start();
              consQ.start();
              consR.start();
              consS.start();
       }
    }
```

The following line creates the `BlockingQueue`:

```
BlockingQueue<String> blkQue = new LinkedBlockingQueue<String>();
```

And it shows how it deploys the `LinkedBlockingQueue` implementation of the `BlockingQueue`. As the `BlockingQueue` is an abstract class, it is not possible to directly instantiate it. Another option could be to use the queue type, `ArrayBlockingQueue`. The advantage of using it is that it uses an array to store data, whereas the `LinkedBlockingQueue` class uses a linked list. An `ArrayBlockingQueue` has fixed size. However, in case of a `LinkedBlockingQueue`, you can specify the maximum size, which is by default unbounded.

During the execution of the class, the dequeuing of objects from the queue happens in a sequential order, as shown in Figure 19.23:

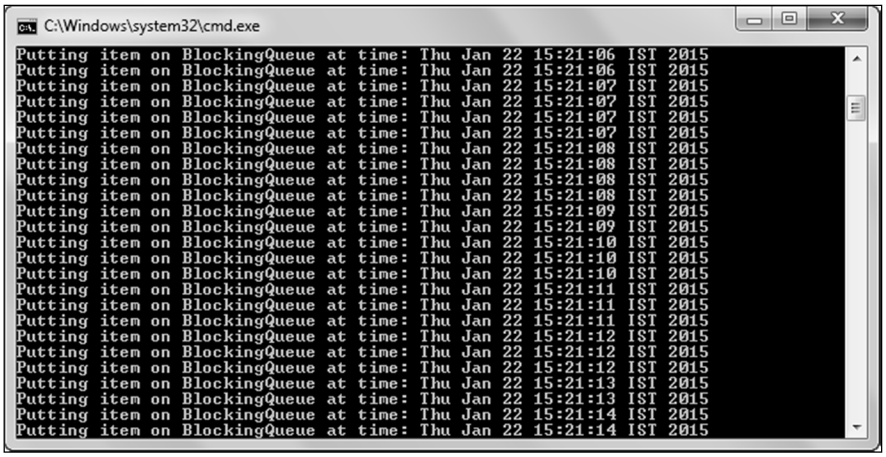


Figure 19.23: Consumer Threads

Further, other consumer threads are blocked by one consumer thread from accessing the `BlockingQueue` till the time it is able to dequeue an object from the queue. Messages are dequeued from the `BlockingQueue` by these threads in the same order as you spawned them.

Implementing Concurrent Programming

There is a built-in language support for threads in Java, and they are used to handle programs in an easy manner. Some of the built-in primitives, such as synchronized blocks, `Object.wait()`, and `Object.notify()`, are not enough to perform many programming tasks.

With J2SE 5, a new path to multithreading was provided in the Java programming language. The real processes for coordinating threads with `wait()` and `notify()` are now elaborated with new and sophisticated mechanisms for working with threads. Here, we will focus on concurrency mechanisms that are included in Java SE 7.0. The new packages are now a part of the java.util.concurrent package.

Limitations of Synchronization Previous to J2SE 5

While creating multithreaded applications, one of the most difficult problems faced was related to data synchronization. Such errors make design harder and are usually difficult to detect. Built-in synchronization has the following limitations:

- No option to withdraw an operation of trying to acquire a lock that is already held or to give up after waiting for a specific period of time or to cancel an attempt to lock after an interrupt
- No option provided to alter the semantics of a lock, such as, with regard to re-entrancy, read as opposed to write protection, or fairness
- No access control for synchronization. Any method can perform synchronized (`res`) for any accessible resource
- Synchronization is possible only inside methods and blocks. It implies that a lock cannot be acquired in one method and released in another

These problems can be overcome with the help of the utility classes that control locking, mutex for instance, which is another term used for a lock. However, a mutex never nests the way synchronization methods or blocks do, as shown here:

```
public class Mutex
{
  public void acq() throws InterruptedException { }
  public void rel() { }
  public boolean attmpt(long msec) throws InterruptedException { }
}
```

and it can be used as:

```
try { mutex.acq();
try {

}
finally { mutex.rel();
} catch(InterruptedException ie) { }
```

Developers can use the standardized classes that provide building blocks for creating various applications.

Simplifying Servers Using the Concurrency Utilities

Similar to Java SE 7, Java SE 8 too comes with support for multiprogramming in the Java programming language. Originally, threads were coordinated with the help of methods, such as `wait()` and `notify()`. However, they are now enhanced by using advanced mechanisms to work with threads. The `java.util.concurrent` package offers advanced mechanisms that provide several concurrency utilities. These utilities enable the process of developing multithreaded applications and servers easier.

In addition, such standards will improve the quality of such applications. The Java Community Process has defined the package as JSR 166: Concurrent Utilities. The primary goal of the JSR 166 specification is to standardize a framework that brings together commonly used utilities for concurrent programming into a small package, which developers can learn easily.

Its other goals are to facilitate high-quality implementations containing classes and interfaces for atomic variables, locks, barriers, semaphores and condition variables, queues and associated collections created for multithreaded use, and thread pools and a custom execution framework.

Knowing Various Concurrency Utilities

As you have just read, the `java.util.concurrent` package contains classes and interfaces that are used to simplify the process of developing concurrent classes and applications. This package facilitates high-quality implementations of common building blocks that can be used to create concurrent applications. The package contains classes, enhanced to facilitate concurrent access, including:

- **Task scheduling framework**—The Executor framework aims at standardizing tasks such as invocation, scheduling, execution, and control of asynchronous tasks as per execution policies. The processes are permitted to run in a newly created thread or in a pool of threads or in a single background thread. Further, the built-in implementations enable policies that can be configured such as queue length limits and saturation policy. These policies will help in improving the accuracy of applications by avoiding runaway source consumption.
- **Concurrent collections**—Several collection classes have been introduced, which include `Queue` and `BlockingQueue` interfaces, and high-quality concurrent implementations of Map, List, and Queue as well.
- **Atomic variables**—The `java.util.concurrent.atomic` package facilitates developers to automatically calculate values that can be of primitive types or references. The implementation of this package offers high quality performance that can be available through synchronization. This package helps in using high-performance concurrent algorithms, counters, and sequence number generators.
- **Synchronizers**—Synchronization classes that facilitate coordination between threads have been added including semaphores, mutexes, barriers, latches, and exchangers.
- **Locks**—You can use the synchronized keyword for locking purposes. However, as already stated, there are restrictions with this built-in monitor. The `java.util.concurrent.locks` facilitates a high-quality lock implementation and also enables developers to provide a timeout when trying to acquire a lock, multiple condition variables per lock, non-nested scoped locks.
- **Nanosecond-granularity timing**—The introduction of the new `java.lang.System.nanotime()` method facilitates high-precision timing facilities. This method provides access to nanosecond-granularity time source to make relative time measurements and methods, and accept timeouts such as `BlockingQueue.offer()`, `BlockingQueue.poll()`, `Lock.tryLock()`, `Condition.await()`, and `Thread.sleep()`. It should be noted that the precision of `nanotime()` is platform-dependent.

The following are the benefits of using the new `java.util.concurrent` package:

- **Reduced programming effort**—Developers need not have to write new code as they can easily use a standard class.
- **Increased performance**—The implementations are much faster and more scalable as compared to typical implementations because of the fact that they have been developed and tested by concurrency and performance experts.
- **Increased reliability**—While using low-level concurrency primitives, such as synchronized `wait()` and `notify`(), it becomes challenging to correctly use these primitives. Further, errors are also difficult to detect or debug. On the contrary, the concurrency utilities are standardized and comprehensively tested for deadlock, starvation, or race conditions.
- **Improved maintainability**—As applications are based on standardized packages, they become easier to maintain.
- **Increased productivity**—Developers can easily understand the standard library classes and, hence, need not have to learn new APIs.

Learning about the **java.util.concurrent** Package

The `java.util.concurrent` package has already been discussed in the preceding topic. In this section, we discuss the interfaces classes in the package. The interfaces and their description are mentioned in Table 19.6 and the classes in Table 19.7:

Table 19.6: Interfaces of the concurrent package	
Interface	**Does this**
`BlockingDeque<E>`	It is deque that additionally supports blocking operations that wait for the deque to become non-empty when retrieving an element and wait for space to become available in the deque when storing an element
`BlockingQueue<E>`	It is queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element
`Callable<V>`	It is a task that returns a result and may throw an exception
`CompletionService<V>`	It is the service that decouples the production of new asynchronous tasks from the consumption of the results of completed tasks
`ConcurrentMap<K,V>`	It creates a Map providing additional atomic `putIfAbsent()`, `remove()`, and `replace()` methods
`ConcurrentNavigableMap<K,V>`	A `ConcurrentMap` supporting `NavigableMap` operations
`Delayed`	The mix-in style interface for marking objects that should be acted upon after a given delay
`Executor`	An object that executes submitted `Runnable` tasks
`ExecutorService`	It is the executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks
`Future<V>`	The Future represents the result of an asynchronous computation
`RejectedExecutionHandler`	The handler for tasks that cannot be executed by a `ThreadPoolExecutor`
`RunnableFuture<V>`	A Future that is `Runnable`
`RunnableScheduledFuture<V>`	A `ScheduledFuture` that is `Runnable`
`ScheduledExecutorService`	An `ExecutorService` that can schedule commands to run after a given delay, or to execute periodically
`ScheduledFuture<V>`	A delayed result-bearing action that can be cancelled
`ThreadFactory`	An object that creates new threads on demand

Table 19.7: Classes of the concurrent package	
Class	**Does this**
`AbstractExecutorService`	It provides default implementation of `ExecutorService` execution methods
`ArrayBlockingQueue<E>`	A bounded blocking queue backed by an array
`ConcurrentHashMap<K,V>`	A hash table supporting full concurrency of retrievals and adjustable expected concurrency for updates
`ConcurrentLinkedQueue<E>`	An unbounded thread-safe queue based on linked nodes
`ConcurrentSkipListMap<K,V>`	A `ConcurrentNavigableMap` implementation which is scalable
`ConcurrentSkipListSet<E>`	A concurrent `NavigableSet` implementation based on `ConcurrentSkipListMap` which is scalable
`CopyOnWriteArrayList<E>`	A thread-safe variant of `ArrayList` in which all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array.
`CopyOnWriteArraySet<E>`	A set that uses `CopyOnWriteArrayList` for all of its operations
`CountDownLatch`	A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes

Table 19.7: Classes of the concurrent package

Class	Does this
`CyclicBarrier`	A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point
`DelayQueue<E extends Delayed>`	An unbounded blocking queue of Delayed elements, in which an element can only be taken when its delay has expired
`Exchanger<V>`	A synchronization point at which two threads can exchange objects
`ExecutorCompletionService<V>`	A `CompletionService` that uses a supplied Executor to execute tasks
`Executors`	Factory and utility methods for Executor, `ExecutorService`, `ScheduledExecutorService`, `ThreadFactory`, and `Callable` classes defined in this package
`FutureTask<V>`	A cancellable asynchronous computation
`LinkedBlockingDeque<E>`	A blocking deque based on linked nodes which is optionally bounded
`LinkedBlockingQueue<E>`	An optionally-bounded blocking queue based on linked nodes
`PriorityBlockingQueue<E>`	An unbounded blocking queue that uses the same ordering rules as class `PriorityQueue` and supplies blocking retrieval operations
`ScheduledThreadPoolExecutor`	A `ThreadPoolExecutor` that can additionally schedule commands to run after a given delay, or to execute periodically
`Semaphore`	A counting semaphore
`SynchronousQueue<E>`	A blocking queue in which each put must wait for a take, and vice versa
`ThreadPoolExecutor`	An `ExecutorService` that executes each submitted task using one of possibly several pooled threads, normally configured using Executors factory methods
`ThreadPoolExecutor.AbortPolicy`	A handler for rejected tasks that throws a `RejectedExecutionException`
`ThreadPoolExecutor.CallerRunsPolicy`	A handler for rejected tasks that runs the rejected task directly in the calling thread of the `execute` method, unless the executor has been shut down, in which case the task is discarded
`ThreadPoolExecutor.DiscardOldestPolicy`	A handler for rejected tasks that discards the oldest unhandled request and then retries execute, unless the executor is shut down, in which case the task is discarded
`ThreadPoolExecutor.DiscardPolicy`	A handler for rejected tasks that silently discards the rejected task

Learning about the **java.util.concurrent.locks** Package

The `java.util.concurrent.locks` provides classes for locking and waiting for conditions that are distinct from built-in synchronization and monitors. Even if this framework has awkward syntax, it brings greater flexibility when using locks and conditions. The interfaces and classes of this package are mentioned in the tables that follow. This package provides reader/writer locks.

Reader/Writer Locks

When using a thread to read data from an object, you do not need to prevent another thread from reading data at the same time. As long as threads are just reading the data and not making any modifications to it, there is absolutely no reason to lock threads. The `java.util.concurrent.locks` package provides classes that implement this type of locking.

The `ReadWriteLock` interface provides locks for read-only and for writing. Multiple reader threads can use `readLock()` simultaneously till the time there are no writers. The `writeLock()` is exclusive. Using

reader/writer locks increases concurrency and results in better performance, rather than using a mutual exclusion lock. However, such an improvement in the performance can be benefitted only on a multi-processor.

The interfaces and their description are mentioned in Table 19.8 and the classes in Table 19.9:

Table 19.8: Interfaces of the java.util.concurrent.locks package

Interface	Does this
`Condition`	Condition factors out the Object monitor methods (wait, notify and `notifyAll`) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary Lock implementations.
`Lock`	Lock implementations provide more extensive locking operations than can be obtained using synchronized methods and statements.
`ReadWriteLock`	A `ReadWriteLock` maintains a pair of associated locks, one for read-only operations and one for writing.

Table 19.9: Classes of java.util.concurrent.locks package

Class	Does this
`AbstractOwnableSynchronizer`	A synchronizer which may be exclusively owned by a thread
`AbstractQueuedLongSynchronizer`	An `AbstractQueuedSynchronizer` maintaining synchronization state as long.
`AbstractQueuedSynchronizer`	It provides a framework for implementing blocking locks and related synchronizers (semaphores, events, etc.) that rely on FIFO wait queues.
`LockSupport`	It is a basic thread blocking primitives for creating locks and other synchronization classes.
`ReentrantLock`	A re-entrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities
`ReentrantReadWriteLock`	An implementation of `ReadWriteLock` supporting similar semantics to `ReentrantLock`
`ReentrantReadWriteLock.ReadLock`	The lock returned by method `ReentrantReadWriteLock.readLock()`
`ReentrantReadWriteLock.WriteLock`	The lock returned by method `ReentrantReadWriteLock.writeLock()`

Learning about the **java.util.concurrent.atomic** Package

The `java.util.concurrent.atomic` package provides classes that support lock-free thread-safe programing on single variables. The classes here extend the notion of volatile values, fields, and array elements. The classes of this package are listed in Table 19.10:

Table 19.10: Classes of the java.util.concurrent.atomic package

Class	Does this
`AtomicBoolean`	The `boolean` value that may be updated atomically
`AtomicInteger`	An `int` value that may be updated atomically
`AtomicIntegerArray`	An `int` array in which elements may be updated atomically
`AtomicIntegerFieldUpdater<T>`	The reflection-based utility that enables atomic updates to designated volatile `int` fields of designated classes
`AtomicLong`	A `long` value that may be updated atomically
`AtomicLongArray`	A `long` array in which elements may be updated atomically

Table 19.10: Classes of the java.util.concurrent.atomic package	
Class	**Does this**
`AtomicLongFieldUpdater<T>`	A reflection-based utility that enables `atomic` updates to designated volatile long fields of designated classes
`AtomicMarkableReference<V>`	An `AtomicMarkableReference` maintains an object reference along with a mark bit, that can be updated atomically
`AtomicReference<V>`	An object reference that may be updated atomically
`AtomicReferenceArray<E>`	An array of object references in which elements may be updated atomically
`AtomicReferenceFieldUpdater<T,V>`	A reflection-based utility that enables atomic updates to designated volatile reference fields of designated classes
`AtomicStampedReference<V>`	An `AtomicStampedReference` maintains an object reference along with an integer "stamp", that can be updated atomically

Multiple operations are treated as atomic if using the classes that belong to this package. For instance, you cannot use a volatile integer with the ++ operator as this operator has several instructions.

The `AtomicInteger` class contains a method using which the integer it holds can be incremented atomically without synchronization. For complicated tasks, atomic classes can be used. These tasks include writing code that needs no synchronization. The following code is a sample sequence number generator that uses the `AtomicLong` class:

```
import java.util.concurrent.atomic.*;
public class SampleSequenceGenerator
{
  private AtomicLong number = new AtomicLong(0);
  public long next()
  {
        return number.getAndIncrement();
  }
}
```

Objects belonging to the classes `AtomicBoolean`, `AtomicInteger`, `AtomicLong`, and `AtomicReference` facilitate access and updates to a single variable of the corresponding type. Furthermore, all these classes have their own utility methods for that type, e.g., atomic increment methods.

These classes also have the `getAndSet()` method that assigns a new value to the variable and returns the previous value without using any synchronization locks. Methods such as `compareAndSet()` and `weakCompareAndSet()` are conditional modifier methods that take two arguments. The first argument takes the original value when the method starts, and the second argument takes a new value to assign the data to.

The `java.util.concurrent` package facilitates concurrency utilities. There are no new features added to the classes that belong to this package; however, these classes are more efficient to use and provide more quality implementations for data synchronization techniques. This package has provided all the functionalities for concurrent programming what the collection framework has provided for data structures, thereby refraining the developer from making any new inventions that might possibly lead to incorrect implementations.

The `java.util.concurrent` package also enables you to create applications that are shorter, cleaner, more reliable, easy to read and write, and easy to maintain.

If developers are using this package, it does not mean that their task has reduced. They will still have to create applications that should be able to handle any synchronization issues.

There can be overlap between the classes. For instance, a semaphore having a single permit can be utilized to hold a lock. Therefore, prior to using a new package in an efficient and protected manner, there are several details that must be explored and clearly understood. It is always a good practice to review the packages and their classes before actually using them.

Summary

In this chapter, you have read about the Threads, including creating a thread and naming a thread. You have learned how to use the `Runnable` interface. Further on, there was a comprehensive discussion on Double Buffering, Synchronization, and the `java.util.concurrent` Package.

In the next chapter, you'll read about performing Networking and Security with Java.



20

Networking and Security with Java

If you need an immediate solution to:	*See page:*
Understanding Networking Interfaces and Classes in the java.net Package	734
Understanding the InetAddresses	736
Caching InetAddress	736
Creating and Using Sockets	740
Creating TCP Clients and Servers	743
Understanding the Whois Example	746
Submitting an HTML Form from a Java Program	748
Handling URL	749
Using the URLConnection Objects	752
Working with Datagrams	756
Datagrams Server and Client	758
Working with BART	759
Learning about the java.security Package	761

In Depth

Networking is a particularly important topic in Java. In this chapter, we'll take a look at the `URL` class, which is how about Java handles URLs, the `URLConnection` class, which allows you to get information about Web pages (including their content), and the `Socket` class, which you use to create sockets for communication across the Internet and intranets. The `Socket` class is particularly powerful—using it, you can write client/server Internet programs in which the client is on one machine and the server is on another.

In this chapter, you'll also come across with Security concepts in Java. These concepts basically tell you about the various classes and policies required for providing security in Java.

As we all know that Java, is much more appropriate as the networking language than any other language like C++ or FORTRAN for networking, which makes it be called "programming for the Internet" by its creators. In light of the preceding fact, the support provided by the `java.net` package for networking will be the focus of this chapter.

Basics of Networking

The systems connected over a network through the Internet connect to each other with the help of Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). In Java, programming is done on the application layer that uses the concept of classes in the `java.net` package; these classes provide a system-independent network.

Consider an example of communication that occurs between two ends, such as a telephone conversation. TCP ensures that the data send from one end to other goes in the same order as it was sent. Otherwise, error occurs.

The examples of applications that need a consistent communication network are Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Telnet. The success of these applications depends on the sequence in which the data is set and received over the network. To read the data from the Uniform Resource Locator (URL), the data must be received in the same order it was sent. Otherwise, a corrupt zip file or an HTML file is retrieved in the end.

The UDP is connectionless unlike TCP. It sends datagrams (independent packets of data) from one application to another. Here, there is no significance of the order of delivery and cannot be guaranteed because each message is not related to the other.

Basically, a computer has a single physical connection with a network, and all the other devices are associated with it through this network. The data transmitted over the network is identified by its address that consists of IP address and port number.

Ports are specified by 16-bit numbers and ranged from 0 to 65,535. However, the port numbers that are ranging from 0 to 1023 are reserved to be used by the protocols, such as HTTP and FTP, and other services of system. These ports are known as well-known ports.

In case of datagram-based communication, such as UDP, the port number of destination is contained by the datagram, packet and UDP transfers the packet to the destination accordingly. On the other hand, in case of connection-based communication, such as TCP, a socket with a particular port number encapsulates a server application. This allows the registration of the server with the system to receive all data efficiently.

Java programs use the TCP or UDP protocols for communicating over the Internet by using the classes in java.net. The URL, `URLConnection`, `Socket`, and `ServerSocket` classes, all use TCP for communicating over the network. The `DatagramPacket`, `DatagramSocket`, and `MulticastSocket` classes use UDP for communicating over the network.

Networking includes two types of programs—client programs and server programs. A server program is a program that provides services to one or more users who run client programs to access those services. These client and server computers interact with each other through well-established protocols, which define the nature of the communications between clients and servers.

Networking Enhancements in Java SE 8

With the evolution of Java SE 8, some additions have been made in the networking feature of Java. The various networking enhancements in Java SE 8 are as follows:

- Addition of a new class `java.net.URLPermission`. The is used for providing permission for accessing a resource (or collection of resources) defined by a given URL, and for a given set of request methods and request headers which can be set by a user.
- Addition of the `jdk.net` package which contains socket options that are platform specific and a way to set these options on all of the standard socket types. The definition of socket options is provided in the `jdk.net.ExtendedSocketOptions` package. The first socket option is defined with the name SO_FLOW_SLA for creating socket flows for multi-tenant applications. The way of setting and getting these extended options is provided by the class `jdk.net.Sockets`.

NOTE

In a multi-tenancy architecture, a software application's single instance is used for serving multiple customers.

- Changes in the class HttpURLConnection. When a security manager is installed in this class, a method is called, which will try to open a connection. For this, the caller must provide one of the following conditions:
  - a "connect" SocketPermission to the combination of host/port of the destination URL
  - a URLPermission that allows this request

In case automatic redirection is enabled, this request gets redirected to the other destination. But the caller must have the permission for connecting to the host/URL that is redirected.

JDK 8 Security Enhancements

The various features and enhancements in JDK 8 release are as follows:

- **Enabled TLS 1.1 and TLS 1.2 protocols**—The TLS 1.1 and TLS 1.2 protocols are enabled by the SunJSSE provider by default on the client.
- Limited doPrivileged—A version of the method AccessController.doPrivileged is added, which allows code to emphasize on a subset of its permissions or privilege, without restricting full searching of the stack for checking other permissions.
- **Addition of stronger algorithms for encryption of password**—Algorithms such as PBEWithSHA256AndAES_128 and PBEWithSHA512AndAES_256 are added to the SunJCE provider for better encryption of password.
- **SSL/TLS Server Name Indication (SNI) extension support in JSSE server**—The feature of SNI extension extends the SSL/TLS protocols. It is done to specify with what server name the client is trying to connect while handshaking. By default, the SNI extension for client applications has been enabled by SunJSSE in JDK 7. However, JDK 8 supports the SNI extension only for the server applications.
- **Support AEAD algorithms**—The enhancement is done in SunJCE provider for supporting AES/GCM/NoPadding cipher implementation along with GCM algorithm parameters. The SunJSSE provider is also enhanced for supporting AEAD mode-based cipher suites.
- **Enhancements in KeyStore**—For accepting a password and storing it securely as a secret key, the importpassword command option is added to the keytool utility. Moreover, a new class with the name java.security.DomainLoadStoreParameter is added for supporting DKS keystore type. More number of new classes such as java.security.PKCS12Attribute are added and other security APIs gets updated for supporting enhancements in KeyStore.
- **SHA-224 Message digests**—The enhancement of cryptographic algorithms is done in JDK 8 with the SHA-224 variant of the SHA-2 family of message-digest implementations.
- **Enhanced support for NSA suite B cryptography**—The enhancement done in NSA suite B cryptography includes the following:
  - OID registration for NSA Suite B cryptography algorithms

- Support for generation of 2048-bit DSA key pair and additional signature algorithms for 2048-bit DSA keys such as SHA224withDSA and SHA256withDSA
- Keysize limitation pncrease from 1024 to 2048 for Diffie-Hellman (DH)

- **Better support for high entropy random number generation**—The SecureRandom class allows cryptographically strong random numbers to be generated for private or public keys, ciphers, signed messages, etc. The getInstanceStrong() method is provided in JDK 8 for returning an instance of the strongest SecureRandom implementation. Other changes done in the SecureRandom class are as follows:
  - For UNIX-like platforms, two new implementations NativePRNGBlocking and NativePRNGNonBlocking were introduced, which provide blocking and non-blocking, behavior, respectively
  - SHA1PRNG and NativePRNG are fixed to follow the SecureRandom seed source properties in the java.security file
- **Addition of a new PKIXRevocationChecker class**—The PKIXRevocationChecker class uses the PKIX algorithm for verifying the cancellation status of certificates. Moreover, the PKIXRevocationChecker class supports end-entity certificate checking, best-effort checking, and mechanism-specific options and parameters.
- **64-bit PKCS11 for Windows**—In JDK 8, the PKCS 11 provider support has been enhanced for Windows from 32 bit to include 64 bit.
- **New rcache types in Kerberos 5 replay caching**—Kerberos 5 includes two new rcache types such as none and dfl. The type none signifies no rcache at all, whereas the type dfl signifies the DFL style file-based rcache. Moreover, the acceptor-requested subkey is also supported. The rcache and subkey are configured by using the sun.security.krb5.rcache and sun.security.krb5.acceptor.subkey system properties.
- **Protocol transition and constrained delegation for Kerberos 5**—JDK 8 supports Kerberos 5 protocol transition and constrained delegation.
- **Weak encryption disabled by default**—By default, the DES-related Kerberos 5 encryption types are not supported, but can be enabled by adding the allow_weak_crypto=true in the krb5.conf file.
- **Unbound SASL for the GSS-API/Kerberos 5 mechanism**—The Krb5LoginModule principal value in a JAAS configuration file is set to asterisk (*) on the acceptor side for denoting an unbound acceptor. This means the server can be accessed by the initiator using any service principal name if the acceptor includes the long-term secret keys to that service. Using the GSSContext.getTargName() method, the name can be accessed by the acceptor after establishing the context.
- **SASL service for multiple host names**—While building a SASL server, you can set the server name to null for denoting an unbound server; therefore, a service can be requested by a client with the help of any server name. After establishing a context, the server can access the name as a negotiated property by using the key name SASL.BOUND_SERVER_NAME.
- **JNI bridge to native JGSS on Mac OS X**—JNI bridge to native JGSS is now supported on Mac OS X. You can enable it by setting the system property sun.security.jgss.native to true.
- **Support stronger ephemeral DH keys in the SunJSSE provider**—During the handshaking of SSL/TLS in the SunJSSE provider, make ephemeral DH key match with the length of the certificate key. A new system property, jdk.tls.ephemeralDHKeySize, is defined for customizing the ephemeral DH key sizes. The minimum acceptable DH key size is 1024 bits, except for exportable cipher suites or legacy mode.
- **Support cipher suites preference customization in JSSE**—The SunJSSE provider allows the preference of client's cipher suite by default. However, you can change the behavior of the server's cipher suite preference by invoking SSLParameters.setUseCipherSuitesOrder(true) in the server.

Sockets in Java

An *IP Port* on a specific host machine is called a *Socket*. The emergence of sockets was first apparent in early Unix systems in the 1970s and have now become the 'standard' low-level communication primitive.

There exists a close resemblance of a network socket with an electrical socket. The plugs have their own standard way of delivering the payload. It is very easy to "plug in" the socket and communicate by anything that understands the standard protocol.

Let's now understand the two types of sockets—*connection-oriented* sockets, almost always based on TCP, and *connectionless* sockets, usually based on *User Datagram Protocol* (UDP). There are quite a few distinguishing differences between the two of them as TCP-type sockets guarantee data arrival and in the correct order, whereas UDP-type ones do not. The major point to note here is that despite the preceding reason, many sophisticated users sometimes use UDP.

TIP

It may appear obvious that 'server sockets execute on servers and client sockets execute on clients.' However, it may not be true all the time- servers can be socket servers, and socket clients, and clients can judicially run server sockets.

Client-Server Networking

The relationship between two computer programs in which one program, the client, makes a service request from another program, the server, which fulfills the request is self-explanatory to discriminate the server from the client. A network architecture comprises clients and servers among which communication occurs. A server is a powerful computer that manages disk drives (*file servers*), printers (*print servers*), or network traffic (*network servers*), whereas clients are those computers on which users run their applications. Clients depend on servers for various resources, such as files, devices, and even processing power.

For the programs running on a network, client server model is a better idea to be implemented compared with a single computer. The client server model is preferred for distributed systems where the systems are at several locations.

An example of cash withdrawal from an ATM machine uses the concept of client server model. The customer requests to the ATM machine for cash from his/her account the ATM machine forwards the request from the user to the bank and after verification (through its database) from the bank, the bank sends approval to the ATM. Further, the ATM responds back by providing the cash according to the instructions given in the field of amount. It automatically generates the receipt if the user asks for it and deducts the amount from the customer's account.

Proxy Servers

In simple words, a proxy server can be viewed as a bridge between the client application, such as a Web browser, and a real server. It gets and gathers all requests from the real server to view if it can satisfy the requests itself. If not, it forward the request to the real server. The following two main goals are served by proxy servers:

- **Improve performance**—To improve the performance of all users, a proxy server is used. A proxy server saves the response for all the requests for a limit of time. Consider an example where both user A and user B access the World Wide Web through a proxy server. User A first requests for a webpage like a website of Java called page 1. After some time, user B also requests for the same page; instead of iterating the request to the server, it invokes page 1 from user A, which was already fetched by user A. It is a time-saving process and proxy servers support hundreds and thousands of users. The major online services that establish an array of proxy servers are Yahoo, America Online, MSN, and many others.
- **Filter requests**—Proxy servers can also be used to filter requests, which means to allow only those requests that are appropriate for the organization. For example, a college might use a proxy server to prevent its students from accessing a specific set of websites.

So we can conclude that a proxy server is a connection between a LAN and an external network such as the Internet. These days, proxy servers are facing competition from NAT and firewalls. In order to remain in demand, the role of proxy servers has been widened to content caching and authentication besides guard-dog security and connection sharing.

Internet Addressing

A computer accessing the Internet has a specific address which is a number that uniquely finds each computer on the Internet. On the Internet, there is only one type of electronic address. Once we know someone's Internet address, we can send mail, transfer files, have a conversation, and even find out information about that person. The word address on the Internet is always referred to as an electronic address which consists of the form—a string followed by a @ character (the "at" sign), which is further followed by the unique name of a computer.

The part of the address mentioned after @ character is known as domain. Each part of a domain is called a sub-domain and they are separated by periods. For example, xyz`@pccrrn.co.uk`.

In the example, there are three sub-domains—pccrrn, co, and uk. You can understand a domain name by looking at the sub-domains, considering them from right to left. The name is created so that each sub-domain notifies the user about the computer. Top-level domain is the rightmost sub-domain. As it is read to the left, the sub-domains become more specific. The top-level domain uk, means the computer is in the United Kingdom. The next sub-domain, co, refers to a company. Next, pccrrn tells the user that is the name of a specific computer.

NOTE

Generally, only lowercase letters are used for Internet addresses. There is no need to mix in uppercase letters. If an address has some of the letters in uppercase, it is always safe to change them to lowercase.

Domain Name Service

The Internet wouldn't be user friendly if users had to refer their intended addresses as numbers. So, a parallel hierarchy of names to go with all these numbers exists. Such a hierarchy is called Domain Name Service (DNS).

Let's understand the concept this way. Host names are created using a naming standard called DNS. If DNS wouldn't have existed, we would be ordering books from 176.32.98.166 instead of `www.amazon.com`; we might sell our CDs at 23.207.146.7 instead of `www.ebay.com` and; we'd search the Web at 173.194.36.84 instead of `www.google.com`.

The DNS server on the Internet resolve hostnames for their corresponding IP addreses. DNS is related to URL, which stands for universal resource locator. URL is a naming scheme that can uniquely identify any resource (file) on the internet.

Inet4Addresses and Inet6Addresses

As we had discussed in the Internet Addressing section that all Internet addresses consist of 32 bit values, organized as four 8-bit values. IP addresses are represented by the dotted-decimal notation. In the dotted-decimal notation, each group of eight bits is called an octet and is specified by its decimal equivalent, such as 192.167.134.12. Besides the IP addresses, a port must be used for accessing data from a computer over the Internet. For example, the port number for the Web protocol HTTP is 80. Port numbers are specified with a colon. For example, the HTTP service on a server whose IP address is 192.167.134.20 is 192.167.143.20:80.

There are three classes provided for IP addressing—`Inetaddress`, `Inet4address`, and `Inet6address`. `Inetaddress` is either a 32- or 128-bit unsigned number used by IP. `Inet4Address` is provided for representing 32-bit `IPv4address and` 128-bit `IPv6addresses`. An `IPv4address` is represented by the form `nnn.nnn.nnn.nnn`, where n is an integer. `Inet4Address` and Inet6Address are the subclasses of InetAddress, as shown in Figure 20.1:

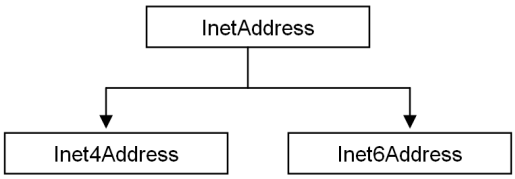


Figure 20.1: The InetAddress

There are two classes for socket addressing, namely, `SocketAddress` and `InetSocketAddress`, as shown in Figure 20.2:

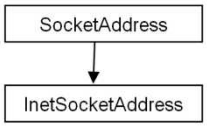


Figure 20.2: The SocketAddress and its class

`SocketAddress` refers to an abstract socket address not dependent on a particular protocol. It is used for sub-classing for a particular protocol. The `InetSocketAddress` class is a subclass of the `SocketAddress` class; it specifies an IP socket address. It can contain an IP address and a port (e.g. 80), or a hostname (e.g., `oracle.com`) and a port (e.g., 2000), or a port only (e.g., 1110). In the latter case, a wildcard IP address is assumed.

The **URL** Class

No browser can identify any information from the Web without using URL. URLs are nothing but Uniform Resource Locator and identify a resource anywhere on the Internet. The advent of URLs with World Wide Web.

Tim Berners-Lee defined a flexible way to locate all the resources of the network. WWW is a light collection of high-level protocols and file formats; URL transforms this loose collection into heavy collection when the information is accurately defined.

The URL is based on four components—protocol (separated from the rest by a colon); host name or IP address of the host to use; port number; and actual file path (most of the HTTP servers will append a file named `index.html`). Some protocols used are http, ftp, gopher, and file. Nowadays, almost everything is being done by HTTP.

For example, `http://docs.oracle.com/javase/8/docs/api/java/net/URL.html` specifies that the protocol used is http (HyperText Transfer Protocol) and that the information is available on a host machine named `docs.oracle.com`. The name of the information on that host machine is `/javase/8/docs/api/java/net/URL.html` and its accurate meaning is dependent on both protocol and host.

A URL can optionally indicate a "port," which is the port number to the TCP connection made on the remote host machine. If the port is not specified, the default port is used. Example: the default port for HTTP is 80.

A relative URL can be specified with sufficient information required to reach the resource relative to another URL. Relative URLs are frequently used within HTML pages. For example, if the content of the URL `http://docs.oracle.com/index.html` contains within it the relative URL, `URL.html`, it would be shorthand for `http://docs.oracle.com/URL.html`.

The **URI** Class

The term URI stands for Uniform Resource Identifier.

This class provides constructors for the URI instances to be created with the help of their components or by parsing (analyzing) their string forms to access the various components of an instance and methods for normalizing and judging URI instances. Instances of this class are immutable, which means that once the object is created, it cannot be modified.

Every URL is a URI, but not every URI is a URL. This is because there is another subdivision of URIs, Uniform Resource Names (URNs), which name the resources but do not specify their location. The news and ISBN URIs are examples of URNs.

A URL class's instance specifies the syntactic components of a URL along with some of the information needed for accessing the resource that is described by it.

A URL must always define a scheme, that is, it must be absolute. A URL string follows this scheme for parsing. A stream handler is necessary for URL scheme, and without this, it is impossible to create a URL instance. A

URL manages the syntactic operation of resolution as well as the I/O functions of looking up for the host and setting up the connection to the resource.

URI Syntax and Components

At the highest level, a URI reference (called URI) in string has a syntax [scheme:] scheme-specific-part [#fragment], where square brackets [..] denote optional components and the symbols : and # denote for themselves. An opaque URI is referred to as an absolute URI and its scheme-specific portion does not initiate with a slash symbol `('/')`. Opaque URIs are not further parsed. The examples for opaque URIs are:

```
mailto:java-net@oracle.com
mailto:java-net@java.sunrise.com
news:comp.lang.java
Urn: isbn:087149110x
```

A hierarchical URI is either an absolute URI or a relative URI. In absolute URI, the scheme-specific part of hierarchical URI starts with a slash character, whereas in case of relative URI, the scheme is not specified. An instance of hierarchical URIs is as follows:

```
http://www.oracle.com/technetwork/java/javase/8-compatibility-guide-2156366.html
```

A URI component has nine components, as shown in Table 20.1:

Table 20.1: The component and its type of URI Instance

Component	Type
Scheme	String
Scheme-specific-part	String
Authority	String
User-info	String
Host	String
Port	int
Path	String
Query	String
Fragment	String

At any given instance, a particular component remains undefined or defined by using a unique value. The former are represented by null, while later are represented by -1. A `String` component is defined with the empty string as its value.

The type of URI being represented decides whether any particular component would be defined or not. A scheme component is present in an absolute URI. An opaque URI comprises a scheme-specific part and a fragment, but contains no other components. A hierarchical URI always comprises a path that can be empty and a scheme-specific-part that might contain any of the other components.

TCP/IP and Datagram

TCP/IP provides serialized, expected, reliable, and flexible stream of packet data. But it comes with a cost. TCP includes many complications. One of the complications is that once the datagram gets released to its target, there is no guarantee that the datagram will arrive or someone is there for catching it. Similarly, when the datagram is received, it could be damaged as well.

Blackboard Assignment Retrieval Transaction

Bart, short for Blackboard Assignment Retrieval Transaction, is named such because it enables blackboard assignments to be retrieved randomly.

On calling the server, `BartS`, the protocol (Bart) used by the client, `BartC`, to communicate with the Bart server is called because it allows you to randomly retrieve blackboard assignments. Bart server, `BartS`, communicates over port 1234. A supporting class is also required that helps in network programming. This class, `BartQ`, has a single method named `getQuote()` that returns one of the 20 randomly selected blackboard sentences.

```
import java.util.ArrayList;
public class BartQuote {
  ArrayList<String> q = new ArrayList<String>();
  public BartQuote() {
        q.add("I will not waste chalk.");
        q.add("I will not skateboard in the halls.");
        q.add("I will not burp in class.");
        q.add("I will not instigate a revolution.");
        q.add("It's potato, not potatoe.");
        q.add("I will not encourage others to fly.");
        q.add("Tar is not a plaything.");
        q.add("I will not sell school property.");
        q.add("I will not get very far with this attitude.");
        q.add("I will not sell land in Florida.");
        q.add("I will not grease the monkey bars.");
        q.add("I will not hide behind the Fifth Amendment.");
        q.add("I am not a dentist.");
        q.add("I will finish what I sta");
        q.add("Hamsters cannot fly.");
        q.add("I will not aim for the head.");
        q.add("I will not expose the ignorance of the faculty.");
        q.add("I will not conduct my own fire drills.");
        q.add("I will not fake seizures.");
        q.add("This punishment is not boring and meaningless.");
  }
        public String getQuote() {
        int i = (int)(Math.random() * q.size());
        return q.get(i);
  }
}
```

Just compile this program. We'll look at a full-fledged program that shows us the communication between the server and client using the class `BartQuote` in the "Immediate Solutions" section.

Immediate Solutions

Understanding Networking Interfaces and Classes in the java.net Package

Now let's understand how Java relates to all the network concepts. "OK", says the NP. "So does Java support the functionality of networks?" You smile and say, "Java makes it easy to connect to other computers using classes from the `java.net` package."

Java supports TCP/IP suite for networking by enhancing the already created I/O interface and adding the required features for creating I/O objects. Java supports both TCP and UDP protocols. The former is used for stream-based I/O across the network, and the latter supports point-to-point transfer of datagrams.

The Networking Interfaces and Classes

The interfaces of the `java.net` package can be found in Table 20.2 and classes in Table 20.3:

Table 20.2: Interfaces of the java.net package

Interface	Description
`ContentHandlerFactory`	This interface defines a factory for content handlers
`CookiePolicy`	Its implementation decides which cookies should be accepted and which should be rejected
`CookieStore`	It represents a storage for cookie
`DatagramSocketImplFactory`	This interface defines a factory for datagram socket implementations
`FileNameMap`	A simple interface, which provides a mechanism to map between a file name and a MIME type string
`SocketImplFactory`	This interface defines a factory for socket implementations
`SocketOptions`	Interface of methods to get/set socket options
`URLStreamHandlerFactory`	This interface defines a factory for URL stream protocol handlers

Table 20.3: Classes of the java.net package

Class	Description
`Authenticator`	The class `Authenticator` represents an object that knows how to obtain authentication for a network connection
`CacheRequest`	It represents channels for storing resources in the `ResponseCache`
`CacheResponse`	It represents channels for retrieving resources from the `ResponseCache`
`ContentHandler`	The abstract class `ContentHandler` is the `superclass` of all classes that read an Object from a `URLConnection`
`CookieHandler`	A `CookieHandler` object provides a `callback` mechanism to hook up a HTTP state management policy implementation into the HTTP protocol handler
`CookieManager`	A class providing concrete implementation of `CookieHandler`
`Datagram Packet`	This class represents a datagram packet
`Datagram Socket`	This class represents a socket for sending and receiving datagram packets
`DatagramSocketImpl`	It is abstract datagram and multicast socket implementation base class
`HttpURLConnection`	A `URLConnection` with support for HTTP-specific features

Table 20.3: Classes of the java.net package

Class	Description
`IDN`	This class has methods to convert Internationalized Domain Names (IDNs) between a normal Unicode representation and an ASCII Compatible Encoding (ACE) representation
`Inet4Address`	This class represents an Internet Protocol version 4 (IPv4) address
`Inet6Address`	This class represents an Internet Protocol version 6 (IPv6) address
`InetAddress`	This class represents an Internet Protocol (IP) address
`InetSocketAddress`	This class implements an IP Socket Address (IP address + port number). It can also be a pair (hostname + port number), in which case an attempt will be made to resolve the hostname
`InterfaceAddress`	It represents a Network Interface address
`JarURLConnection`	A `URLConnection` to a Java ARchive (JAR) file or an entry in a JAR file
`MulticastSocket`	The multicast datagram socket class is useful for sending and receiving IP multicast packets
`NetPermission`	This class is for various network permissions
`NetworkInterface`	This class represents a Network Interface made up of a name, and a list of IP addresses assigned to this interface
`PasswordAuthentication`	The class `PasswordAuthentication` is a data holder that is used by Authenticator
`Proxy`	This class represents a proxy setting, typically a type (http, socks) and a socket address
`ProxySelector`	It selects the proxy server to use, if any, when connecting to the network resource referenced by a URL
`ResponseCache`	It represents implementations of `URLConnection` caches
`SecureCacheResponse`	It represents a cache response originally retrieved through secure means, such as TLS
`ServerSocket`	This class implements server sockets
`Socket`	This class implements client sockets (also called just "sockets")
`SocketAddress`	This class represents a `SocketAddress` with no protocol attachment
`SocketImpl`	The abstract class `SocketImpl` is a common superclass of all classes that actually implement sockets
`SocketPermission`	This class represents access to a network via sockets
`URI`	It represents a Uniform Resource Identifier (URI) reference or an Internationalized Resource Identifier (IRI) reference
`URL`	Class URL represents a Uniform Resource Locator, a pointer to a "resource" on the World Wide Web
`URLClassLoader`	This class loader is used to load classes and resources from a search path of URLs referring to both JAR files and directories
`URLConnection`	The abstract class `URLConnection` is the superclass of all classes that represent a communications link between the application and a URL
`URLDecoder`	It is utility class for HTML form decoding
`URLEncoder`	It is utility class for HTML form encoding
`URLStreamHandler`	The abstract class `URLStreamHandler` is the common superclass for all stream protocol handlers

Understanding the **InetAddresses**

The platform on which TCP and UDP are built is called Internet Protocol (IP) address. An IP address comprises either a 32-bit or 128-bit unsigned number used by IP and uses host name with which either it may be constructed or have done a reverse host name resolution. An IP address specifies a number used for identifying a particular host computer on the Internet. Going by that logic, the designers of Java could have simply represented IP addresses as long numbers. The various address types are as follows:

- **Unicast**—As the name indicates, it is an identifier for a single interface. The unicast address is of the following types:
  - **The Unspecified Address**—It is also known as any local or wildcard, it has no address. Basically, it is not assigned to any node. Example of its use is as the target of bind, which allows a server to create a client connection on any interface in case the server host has multiple interfaces. You must avoid using the unspecified address as the destination address of an IP packet.
  - **The Loopback Addresses**—This is the address assigned to the loopback interface. Data received at this IP address circulates around and gets IP input on the local host. This address is often used when testing a client.
- **Multicast**—Multicast refers to an identifier for a collection of interfaces. A packet provided to a multicast address gets delivered to all interfaces determined by that address.

IP Addresses Scope

The IP address scope can be divided into:

- **Link-local addresses**—These addresses are developed to address a single link for various requirements such as configuration of auto-address for neighbor discovery.
- **Site-local addresses**—These addresses are developed for addressing inside of a site despite the requirement for a global prefix.
- **Global addresses**—They are distinct across the Internet.

Host Name Resolution

The resolution of host name-to-IP address can be achieved by using the combination of configuration information of local machine and network naming services such as DNS and Network Information Service (NIS). The specific naming services(s) used by default is the local machine configured. For any host name, its corresponding IP address is returned.

Reverse name resolution specifies returning of host names for any IP address related to it. The methods for resolving host names to their IP addresses and vice versa are provided by the `InetAddress` class.

Caching **InetAddress**

The `InetAddress` class provides the efficiency of caching successful as well as unsuccessful resolutions of host name. DNS spoofing attacks are guarded by the positive caching, while negative caching improves the performance of the system, but for a very short period of time.

Under some conditions, where it can be predicted that DNS spoofing attacks (falsifying the data) are not possible, a Java security property can be set to a different Time-To-Live value for positive caching. Similarly, an admin can configure negative caching TTL value when required. The two properties of Java security that can control the values of TTL for both positive and negative resolutions of host name are as follows:

- `networkaddress.cache.ttl (default: -1)`—Denotes the policy of caching for successful name lookups from the name service. The value is mentioned as integer that signifies the number of seconds for caching the successful lookup. A value of -1 indicates "cache forever".
- `networkaddress.cache.negative.ttl (default: 10)`—Denotes the policy of caching for unsuccessful name lookups from the name service. The value mentioned as integer signifies the number of seconds for caching the failure of unsuccessful lookups.

TIP

A value of 0 indicates "never cache". A value of -1 indicates "cache forever".

Table 20.4 describes the methods of the `InetAddress` class:

Table 20.4: Methods of the InetAddress class

Method	Does this
`boolean equals(Object obj)`	It compares this object against the specified object
`byte[] getAddress()`	It returns the raw IP address of this `InetAddress` object
`static InetAddress[] getAllByName(String host)`	It is given the name of a host, returns an array of its IP addresses, based on the configured name service on the system
`static InetAddress getByAddress(byte[] addr)`	It returns an `InetAddress` object given the raw IP address
`static InetAddress getByAddress(String host, byte[] addr)`	It creates an `InetAddress` based on the provided host name and IP address No name service is checked for the validity of the address
`static InetAddress getByName(String host)`	It determines the IP address of a host, given the host's name
`String getCanonicalHostName()`	It gets the fully qualified domain name for this IP address
`String getHostAddress()`	It returns the IP address string in textual presentation
`String getHostName()`	It gets the host name for this IP address
`static InetAddress getLocalHost()`	It returns the local host
`int hashCode()`	It returns a hashcode for this IP address
`boolean isAnyLocalAddress()`	It is utility routine to check if the `InetAddress` in a wildcard address
`boolean isLinkLocalAddress()`	It is utility routine to check if the `InetAddress` is an link local address
`boolean isLoopbackAddress()`	It is utility routine to check if the `InetAddress` is a loopback address
`boolean isMCGlobal()`	It is utility routine to check if the multicast address has global scope
`boolean isMCLinkLocal()`	It is utility routine to check if the multicast address has link scope
`boolean isMCNodeLocal()`	It is utility routine to check if the multicast address has node scope
`boolean isMCOrgLocal()`	It is utility routine to check if the multicast address has organization scope.
`boolean isMCSiteLocal()`	It is utility routine to check if the multicast address has site scope
`boolean isMulticastAddress()`	It is utility routine to check if the `InetAddress` is an IP multicast address
`boolean isReachable(int timeout)`	It is used to test whether that address is reachable
`boolean isReachable(NetworkInterface netif, int ttl, int timeout)`	It is used to test whether that address is reachable
`boolean isSiteLocalAddress()`	It is utility routine to check if the `InetAddress` is a site local address
`String toString()`	It converts this IP address to a String

Some points about this class are as follows:

- This class does not contain a constructor. Therefore, the standard method of creating it is by calling one of its static methods, such as `getByName()`.
- The `isReachable()`, `getAllByName()`, and `getByName()` methods throw exceptions.
- Several of these methods use the DNS server configured for the system in order to perform DNS queries.
- This class is part of `java.net` package. Thus, any program using it must import either `java.net.InetAddress` or `java.net.*`.

Let's look at this program which prompts the user for a host name and then it looks up the IP addresses for the host name and displays the results. Once done, it asks the user if he or she wants to look up another host:

```
import java.util.Scanner;
import java.net.*;

public class Hosting
{
  static Scanner sc = new Scanner(System.in);
  public static void main(String[] args)
  {
  System.out.println("==========================================");
  System.out.println("Welcome to a website that prompts you for a host name. Enter 1 ,
  it  shows the IP address and asks for more.");
  System.out.println("==========================================");
        String host;
        do {
                System.out.print("\nPlease enter a host name: ");
                host = sc.nextLine();
                try {
                        InetAddress[] addresses =
                        InetAddress.getAllByName(host);
                        for (InetAddress ip : addresses)
                        System.out.println(ip.toString());
                }
                catch (UnknownHostException e)
                {
                        System.out.println("Can't find this host.");
                }
                } while (doAgain());
  }
  private static boolean doAgain()
  {
        System.out.println();
        String s;
        while (true)
        {
                System.out.print("Look up another? (Y or N) ");
                s = sc.nextLine();
                if (s.equalsIgnoreCase("Y")) { return true; }
                else if (s.equalsIgnoreCase("N")) { return false; }
                }
        }
  }
```

The result is shown in Figure 20.3:



Figure 20.3: Searching for IP address from the given host name

Factory() Method

Since the `InetAddress` class has no visible constructors, factory methods have to be used to create an `InetAddress` object. Factory methods are those where static methods in a class return an instance of that class. Instead of overloading a constructor with various parameter lists and cluttering the result, it is better to have unique method names. The commonly used `InetAddress` factory methods are:

```
static InetAddress getbyName(Stringhost)
throws UnknownHostException
static InetAddress getLocalHost()
throws UnknownHostException
static InetAddress[] getAllbyName(Stringhost)
throws UnknownHostException
```

The `InetAddress` representing the local host object is returned by the `getLocalHost()` method. When the host name is passed to the `getByName()` method, it returns an `InetAddress`. An `UnknownHostException` is thrown by these methods if they are unable in finding the name of host.

A single name can be used for representing various machines on the Internet. An array of `InetAddresses` is returned by the `getAllByName()` factory methods. This array displays all of the addresses that a particular name resolves to. An `UnknownHostException` is thrown if it is not able to resolve the name to at least one address.

`InetAddress` also involves the factory method `getByAddress()`, which retrieves an IP address and returns an `InetAddress` object. You can use either an IPv4 or an IPv6 address. In the following example, address and name of the local machine and two well-known Internet websites are printed:

```
import java.net.*;

class InetTest
{

  public static void main(String args[]) throws UnknownHostException
  {
        System.out.println("Printing IP address of Localhost");
        InetAddress Adr = InetAddress.getLocalHost();
        System.out.println(Adr);
        System.out.println("\nPrinting IP address of Computer Deepak");
        Adr = InetAddress.getByName("deepak");

        System.out.println(Adr);
        System.out.println("\nPrinting IP addresses for host
        name www.google.com");

        InetAddress in[] = InetAddress.getAllByName("www.google.com");
        for(int i = 0; i < in.length; i++)
        System.out.println(in[i]);
  }
}
```

It is important to note here that the result shown in Figure 20.4 might differ slightly as per the scenario:

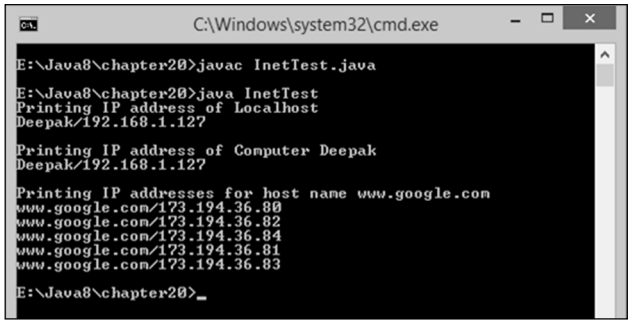


Figure 20.4: Inet addressing

Instance() Method

The `InetAddress` class also contains the `Instance` methods, which you can use on the objects returned by the methods discussed earlier. The most commonly used methods are:

```
boolean equals(Object obj)
byte[] getAddress()
String getHostAddress()
String getHostName()
boolean isMulticastAddress()
String toString()
```

Internet addresses are searched in a hierarchical series of cached servers. In other words, your local host machine may know a specific name-to-IP-address mapping automatically. For other names, it may ask for a local DNS server for IP address data. If that server does not have a particular address, it can go to a remote site and ask for it. This can continue all the way up to the root server (top). This process might take a long time, so it is advisable to structure your code so that you cache IP address information locally rather than viewing it repeatedly.

Creating and Using Sockets

"OK," says the NP, "I want to start working with the Internet up close and personal." "Are you sure?" you ask. "I'm sure," says the NP. "In that case," you say, "check out the `Socket` class."

Communication over the Internet involves sockets for creating connections. Sockets connect to numbered communication ports. The bottom 1,024 ports are reserved for system use (for example, port 21 is for FTP, 23 is for Telnet, 25 is for email, 80 is for HTTP connections, and 119 is for Usenet connections). Java supports sockets with the `Socket` class. Here's the inheritance diagram for the `Socket` class:

```
java.lang.Object
  |____java.net.Socket
```

You'll find the constructors of the `Socket` class in Table 20.5 and its methods in Table 20.6:

Table 20.5: Constructors of the Socket Class

Constructor	Does this
`Socket()`	It creates an unconnected socket with the system-default type of `SocketImpl`
`Socket(InetAddress address, int port)`	It creates a stream socket and connects it to the specified port number at the specified IP address
`Socket(InetAddress host, int port, boolean stream)`	It is deprecated and it uses `DatagramSocket` instead of UDP transport
`Socket(InetAddress address, int port, InetAddress localAddr, int localPort)`	It creates a socket and connects it to the specified remote address on the specified remote port.
`Socket(Proxy proxy)`	It creates an unconnected socket, specifying the type of proxy that should be used regardless of any other settings
`protected Socket(SocketImpl impl)`	It creates an unconnected `Socket` with a user-specified `SocketImpl`
`Socket(String host, int port)`	It creates a stream socket and connects it to the specified port number on the named host
`Socket(String host, int port, boolean stream)`	It is deprecated and it uses `DatagramSocket` instead of UDP transport
`Socket(String host, int port, InetAddress localAddr, int localPort)`	It creates a socket and connects it to the specified remote host on the specified remote port

Table 20.6: Methods of the Socket Class

Method	Does this
`void bind (SocketAddress bindpoint)`	It binds the socket to a local address.
`void close()`	It closes the socket.

Table 20.6: Methods of the Socket Class

Method	Does this
`void connect (SocketAddress endpoint)`	It connects this socket to the server.
`void connect (SocketAddress endpoint, int timeout)`	It connects this socket to the server with a specified timeout value.
`SocketChannel getChannel()`	It returns the unique `SocketChannel` object associated with this socket, if any.
`InetAddress getInetAddress()`	It returns the address to which the socket is connected.
`InputStream getInputStream()`	It returns an input stream for this socket.
`boolean getKeepAlive()`	It tests if `SO_KEEPALIVE` is enabled.
`InetAddress getLocalAddress()`	It gets the local address to which the socket is bound.
`int getLocalPort()`	It returns the local port to which this socket is bound.
`SocketAddress getLocalSocketAddress()`	It returns the address of the endpoint this socket is bound to, or null if it is not bound yet.
`boolean getOOBInline()`	It tests if `OOBINLINE` is enabled.
`OutputStream getOutputStream()`	It returns an output stream for this socket.
`int getPort()`	It returns the remote port to which this socket is connected.
`int getReceiveBufferSize()`	It gets the value of the `SO_RCVBUF` option for this Socket, that is, the buffer size used by the platform for input on this Socket.
`SocketAddress getRemoteSocketAddress()`	It returns the address of the endpoint this socket is connected to, or null if it is unconnected.
`boolean getReuseAddress()`	It tests if `SO_REUSEADDR` is enabled.
`int getSendBufferSize()`	It gets the value of the `SO_SNDBUF` option for this Socket, that is the buffer size used by the platform for output on this Socket.
`int getSoLinger()`	It returns setting for `SO_LINGER`.
`int getSoTimeout()`	It returns setting for `SO_TIMEOUT`.
`boolean getTcpNoDelay()`	It tests if `TCP_NODELAY` is enabled.
`int getTrafficClass()`	It gets traffic class or type-of-service in the IP header for packets sent from this Socket.
`boolean isBound()`	It returns the binding state of the socket.
`boolean isClosed()`	It returns the closed state of the socket.
`boolean isConnected()`	It returns the connection state of the socket.
`boolean isInputShutdown()`	It returns whether the read-half of the socket connection is closed.
`boolean isOutputShutdown()`	It returns whether the write-half of the socket connection is closed.
`void sendUrgentData (int data)`	It sends one byte of urgent data on the socket.
`void setKeepAlive (boolean on)`	It enables/disables `SO_KEEPALIVE`.
`void setOOBInline (boolean on)`	It enables/disables `OOBINLINE` (receipt of TCP urgent data). By default, this option is disabled and TCP urgent data received on a socket is silently discarded.
`void setPerformancePreferences (int connectionTime, int latency,int bandwidth)`	It sets performance preferences for this socket.
`void setReceiveBufferSize (int size)`	It sets the `SO_RCVBUF` option to the specified value for this Socket.
`void setReuseAddress (boolean on)`	It enables/disables the `SO_REUSEADDR` socket option.

Table 20.6: Methods of the Socket Class	
Method	**Does this**
`void setSendBufferSize (int size)`	It sets the `SO_SNDBUF` option to the specified value for this Socket.
`static void setSocketImplFactory (SocketImplFactory fac)`	It sets the client socket implementation factory for the application.
`void setSoLinger (boolean on, int linger)`	It enables/disables `SO_LINGER` with the specified linger time in seconds.
`void setSoTimeout (int timeout)`	It enables/disables `SO_TIMEOUT` with the specified timeout, in milliseconds.
`void setTcpNoDelay (boolean on)`	It enables/disables `TCP_NODELAY` (disable/enable Nagle's algorithm).
`void setTrafficClass (int tc)`	It sets traffic class or type-of-service octet in the IP header for packets sent from this Socket.
`void shutdownInput()`	It places the input stream for this socket at "end of stream."
`void shutdownOutput()`	It disables the output stream for this socket.
`String toString()`	It converts this socket to a String.

Here's an example. In this case, we'll connect to port 80 at `sun.com`, the HTTP port, and get the main Web page at `www.sun.com, index.html` by writing an `HTTP GET` request to that port. To write to a socket, you can use a `PrintWriter` object, and to read from it, you use a `BufferedReader` object. Here's what the code looks like (note that we terminate the HTTP request with two newline characters):

```
import java.io.*;
import java.net.*;
public class SocketDemo
{
  public static void main(String[] args)
  {
        try {
                Socket s = new Socket("www.sun.com", 80);
                BufferedReader in = new BufferedReader(new
                InputStreamReader(s.getInputStream()));
                PrintWriter out = new PrintWriter(s.getOutputStream());
                out.print("GET /index.html\n\n");
                out.flush();
                String line;
                while ((line = in.readLine()) != null) {
                        System.out.println(line); }
        }
        catch (Exception e){ }
  }
}
```

The output of this code is shown in Figure 20.5, which shows reading and writing to a socket:

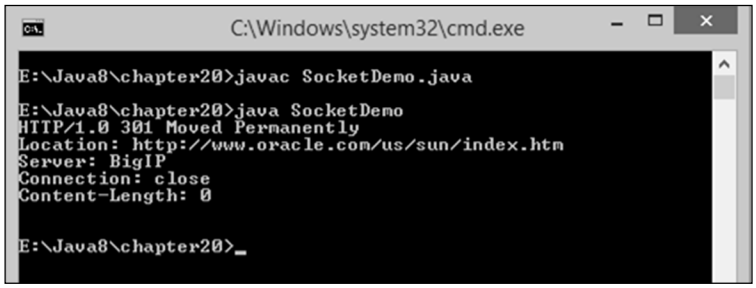


Figure 20.5: Reading and writing to a socket

This example used the Oracle Web server's HTTP port to connect to. However, you can create both sides of the connection—the client and the server—in Java. See the next solution.

Creating TCP Clients and Servers

The Big Boss appears and Says, "We need a secret way of communicating over the Internet for our spies in the field." "We have spies?" You ask. The BB says, "Set up a client/server program that can communicate over the Internet and make sure you scramble everything." You say, "Hmm!"

Using `BufferedReader` and `PrintWriter` objects, you can communicate over the Internet with sockets—all you need is a Domain Name System (DNS) address for the server and a free port on that server (DNS addresses are those groups of four numbers, separated by dots, that specify Internet addresses).

Here's an example. We'll set up a client program that connects to a server program, sends a message, and gets a message back. In this case, we'll use the DNS 127.0.0.1 (the localhost), which means that both the server and the client will be on the same machine. However, we can use any DNS address for a longer time till we put the server script on the machine. We'll use an arbitrary port—number 8765. When you are picking a port number, remember that the port cannot be in use and that both the client and server will connect to the same port. Here's the client application, which sends the message "Dreamtech Press!" to the server and reads what the server sends back:

```
import java.net.*;
import java.io.*;
class ClientCode
{
  public static void main(String args[]) throws Exception
  {
        int character;
        Socket socket = new Socket("127.0.0.1", 8765);
        InputStream in = socket.getInputStream();
        OutputStream out = socket.getOutputStream();
        String string = "Dreamtech Press!\n";
        byte buffer[] = string.getBytes();
        out.write(buffer);
        while ((character = in.read()) != -1)
        {
                System.out.print((char) character);
        }
        socket.close();
  }
}
```

Here's the server application, which reads the message from the client and returns it, after prefacing it with the text "The server got this":

```
import java.io.*;
import java.net.*;

public class ServerCode
{
  public static void main(String[] args )
  {
        try {
                ServerSocket socket = new ServerSocket(8765);
                Socket insocket = socket.accept( );
                BufferedReader in = new BufferedReader
                (new InputStreamReader(insocket.getInputStream()));
                PrintWriter out =
                new PrintWriter(insocket.getOutputStream(), true);
                String instring = in.readLine();
                out.println("The server got this: " + instring);
                insocket.close();
        }
        catch (Exception e) { }
  }
}
```

To run these applications, you'll need two console windows (for example, in Windows, open two DOS windows). Run the server first in one window as shown in Figure 20.6(a) and then the client in the other. When you do, the client reports as shown in Figure 20.6(b):

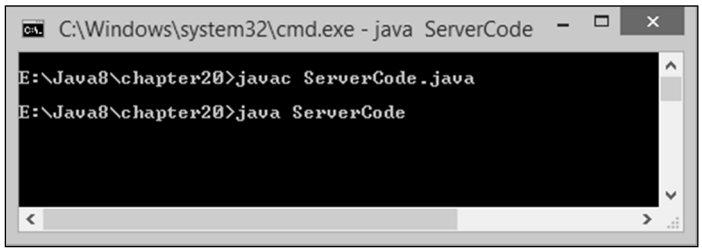


Figure 20.6(a): TCP server window

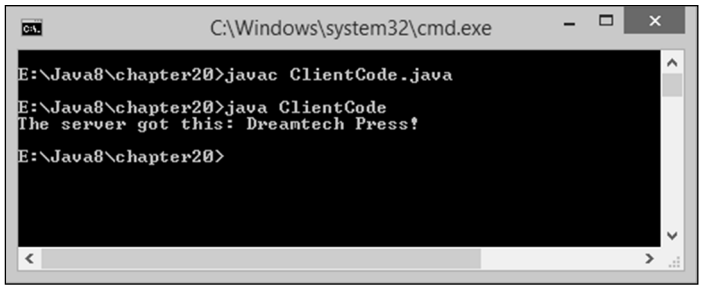


Figure 20.6(b): TCP client window

As you can see, the client sent its message to the server, which sent a confirming message back. Now you are writing Internet client/server applications in Java.

TCP/IP Client Sockets

A socket refers to a special object that is used for establishing connection between the client and server computers. A socket is used for connecting I/O system of Java with other programs that reside either on the local machine or on any other machine on the Internet.

There are two kinds of TCP sockets in Java. One is for server and other is for clients. The `ServerSocket` class is designed to be a "listener", which helps servers establish socket connections with clients. The `Socket` class is designed to connect to server sockets and initiate protocol exchanges.

Let's have an understanding of how to program sockets in Java using the TCP/IP protocol only since it is more widely used than UDP/IP and make sure that `java.net` package is imported when you start off with the program sockets.

To start with the socket programming, you should know how to open a socket. If you are programming a client, then you need to open a socket like this:

```
Socket MyClient;
MyClient = new Socket("My Machine name", PortNumber);
```

In the preceding code snippet, `My Machine name` refers to the machine on which you are attempting for opening a connection and `PortNumber` refers to the port (a number) on which the server you are attempting to connect to is executing. You should select a port number by keeping in mind that port numbers ranging from 0 to 1,023 remain reserved for privileged users. These port numbers remain reserved for some standard services, such as email, FTP, and HTTP. When selecting a port number for your server, the port number greater than 1023 is chosen. Consider the exceptions as well, the preceding can be written as:

```
Socket MyClient;
try { MyClient = new Socket("My Machine name", PortNumber); }
catch (IOException e) { System.out.println(e); }
```

Here arises the need to create an input stream using DataInputStream class to receive response from the server that allows you to read lines of text and Java primitive data types in a portable way by having methods such as read(), readChar(), readInt(), readDouble(), and readLine():

```
DataInputStream inputstr;
try { inputstr = new DataInputStream(MyClient.getInputStream()); }
catch (IOException e) { System.out.println(e); }
```

After creating the input stream, there arises the need to create an output stream on the client side to send information to the server socket using the class PrintStream or DataOutputStream of java.io:

```
PrintStream output;
try { output = new PrintStream(MyClient.getOutputStream()); }
catch (IOException e) { System.out.println(e); }
```

The PrintStream class contains methods that are used to display textual representation of primitive data types. Its Write() and println() methods need to be remembered here. Also, you may want to use the DataOutputStream:

```
DataOutputStream out;
try { out = new DataOutputStream(MyClient.getOutputStream()); }
catch (IOException e) { System.out.println(e); }
```

The class DataOutputStream enables you to create primitive data types of Java and its various methods are used to write a single Java primitive type to the output stream. The method writeBytes() is a useful one.

After creating the input stream and output stream, you need to close the sockets. The output and input streams must always be closed before closing the socket on the client side:

```
try {
output.close();
inputstr.close();
MyClient.close();
}
catch (IOException e) { System.out.println(e); }
```

In this way, you can close the socket after creating the input stream and output stream. There are two constructors which are used to create a Socket object in order to establish a connection between the client and server:

```
Socket(String hostname, int port)
Socket(InetAddress Adr, int port)
```

A socket can be monitored any time for the address and the port information associated with it by using the following methods:

```
InetAddress getInetAddress()
int getPort()
int getLocalPort()
```

With accomplishing this, the access to input and output streams can be easily gained. The getPort() and getLocalPort() throw an IOException if the sockets get invalidated due to the loss of connection on the net. These streams are used exactly like the I/O stream to send and receive the data, like:

```
InputStream getInputStream()
OutputStream getOutputStream()
```

Some points about the Socket class are as follows:

- Even though the Socket class has constructors, the normal way to create a socket is to use the attach() method of the ServerSocket class.
- An object of the OutputStream class is returned by the getOutputStream() method. This is the same class used by System.out. Therefore, the print() and println() methods can be used for sending data to the socket.

Exceptions are thrown by all the constructors and methods of the Socket class. Only the methods getInetAddress(), getPort(), isBound() and isClosed() of the Socket class do not throw exception. Thus, you need to enclose calls to these methods in try/catch statement.

Understanding the **Whois** Example

On the InterNIC server, the open connection to a "whois" port (port43) provides the command-line argument to the socket and then the returned data gets printed. InterNIC may view the argument as a registered Internet domain name, then provide the IP address back and contact information of that site:

```
import java.net.*;
import java.io.*;

class AppWhois
{
  public static void main (String args[]) throws Exception
  {
        int c;
        Socket sokt = null;
        try{
                sokt = new Socket("whois.internic.net", 43, true);
                InputStream inputstr= sokt.getInputStream();
                OutputStream outputstr = sokt.getOutputStream();
                String st = (args.length == 0 ? "hotmail.com" : args[0]) + "\n";
                byte buf[] = st.getBytes();
                outputstr.write(buf);
                while ((c = inputstr.read()) != -1) {
                        System.out.print((char) c); }
        }
        catch(IOException exc){
        System.out.println(exc);
  }
  sokt.close();
}
}
```

If, for example, you obtain information about xxxxxxx site, you'd get something similar to the following:

```
Whois Server Version 2.0

Domain names in the .com and .net domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

HOTMAIL.COM.AU
HOTMAIL.COM.BR
HOTMAIL.COM.IS.HOSTED.ON.PROFITHOSTING.NET
HOTMAIL.COM.IS.N0T.AS.1337.AS.GULLI.COM
HOTMAIL.COM.ZZZZZ.GET.LAID.AT.WWW.SWINGINGCOMMUNITY.COM
HOTMAIL.COM

To single out one record, look it up with "xxx", where xxx is one of the
records displayed above. If the records are the same, look them up
with "=xxx" to receive a full display for each record.
. . .
```

TCP/IP Server Sockets

As we all know that Java has a different Socket class that must be used for creating server applications. You can use the ServerSocket class for building servers that listen either for local or remote client programs for connecting them on published ports. Since the Web is driving most of the activities on the internet, we here develop an operational Web (http) server. If you are programming a server, then this is how you open a socket:

```
ServerSocket MyService;
try { MyService = new ServerSocket(PortNumber); }
catch (IOException e) { System.out.println(e); }
```

A `Socket` object must be created from the `ServerSocket` class for the implementation of the server to listen for connections and accept them from clients:

```
Socket clientSocket = null;
try { serviceSocket = MyService.accept(); }
catch (IOException e) { System.out.println(e); }
```

Once the socket is opened in the server, `DataInputStream` can be used to receive input from the client through the input stream on the server side:

```
DataInputStream input;
try { input = new DataInputStream(serviceSocket.getInputStream()); }
catch (IOException e) { System.out.println(e); }
```

Similarly, in order to create an Output stream on the server side, the class `PrintStream` is used to send information to the client:

```
PrintStream output;
try { output = new PrintStream(serviceSocket.getOutputStream()); }
catch (IOException e) { System.out.println(e); }
```

TIP

You can use the class DataOutputStream as described in the Client Socket.

The important point to note here is that the output and input streams must be closed before closing the socket. To close the socket on the server side:

```
try {
  output.close();
  input.close();
  serviceSocket.close();
  MyService.close();
}
catch (IOException e) { System.out.println(e); }
```

There exists a remarkable difference between a normal socket and a `ServerSocket`. While creating a `ServerSocket`, it registers itself with the system for client connection. The constructors over here simply reflect the port number indicating the length of time you would be in queue before it can refuse or leave the client connection pending. The default is 50. Under adverse conditions, the constructors will throw an `IOException`. Here are three of its constructors:

```
ServerSocket(int port)
ServerSocket(int port, int maxQueue)
ServerSocket(int port, int maxQueue, InetAddress localAddress)
```

`ServerSocket` has a method known as `accept()`, which acts as a blocking call and waits for a client to start communication, and then return with a normal `Socket` that is then used for communication with the client. Table 20.7 shows the list of constructors of ServerSocket and its methods are defined in Table 20.8:

Table 20.7: Constructors of the ServerSocket

Constructor	Does this
`ServerSocket()`	It creates an unbound server socket
`ServerSocket(int port)`	It creates a server socket, bound to the specified port
`ServerSocket(int port, int backlog)`	It creates a server socket and binds it to the specified local port number, with the specified backlog
`ServerSocket(int port, int backlog, InetAddress bindAddr)`	It creates a server with the specified port, listen backlog, and local IP address to bind to

Table 20.8: Methods of the ServerSocket

Method	Does this
`Socket accept()`	It listens for a connection to be made to this socket and accepts it
`void bind (SocketAddress endpoint)`	It binds the `ServerSocket` to a specific address (IP address and port number)

Table 20.8: Methods of the ServerSocket	
Method	**Does this**
`void bind(SocketAddress endpoint, int backlog)`	It binds the `ServerSocket` to a specific address (IP address and port number)
`void close()`	It closes this socket
`ServerSocketChannel getChannel()`	It returns the unique `ServerSocketChannel` object associated with this socket, if any
`InetAddress getInetAddress()`	It returns the local address of this server socket
`int getLocalPort()`	It returns the port on which this socket is listening
`SocketAddress getLocalSocketAddress()`	It returns the address of the endpoint this socket is bound to, or null if it is not bound yet
`int getReceiveBufferSize()`	It gets the value of the `SO_RCVBUF` option for this `ServerSocket`, that is the proposed buffer size that will be used for Sockets accepted from this `ServerSocket`
`boolean getReuseAddress()`	It tests if `SO_REUSEADDR` is enabled
`int getSoTimeout()`	It retrieves setting for `SO_TIMEOUT`
`protected void implAccept (Socket s)`	Subclasses of `ServerSocket` use this method to override `accept()` to return their own subclass of socket
`boolean isBound()`	It returns the binding state of the `ServerSocket`
`boolean isClosed()`	It returns the closed state of the `ServerSocket`
`void setPerformancePreferences(int connectionTime, int latency, int bandwidth)`	It sets performance preferences for this `ServerSocket`
`void setReceiveBufferSize (int size)`	It sets a default proposed value for the SO_RCVBUF option for sockets accepted from this `ServerSocket`
`void setReuseAddress(boolean on)`	It enables/disables the `SO_REUSEADDR` socket option
`static void setSocketFactory (SocketImplFactory fac)`	It sets the server socket implementation factory for the application
`void setSoTimeout (int timeout)`	It enables/disables `SO_TIMEOUT` with the specified timeout, in milliseconds
`String toString()`	It returns the implementation address and implementation port of this socket as a `String`

Submitting an HTML Form from a Java Program

"Java's gone all wacky again!" the NP says. "I have got lots of bells and whistles in my program, but I can't seem to submit an HTML form to the server." "That's because this is Java, not JavaScript," you say, "but there is a way— using sockets."

Many programmers expect that Java has some method of submitting HTML forms, but there's no such method built in. On the other hand, you can create your own technique using sockets and connecting to a Web server's port 80 (the HTTP port). Here's an example in which we submit data to a Perl script, `reg.cgi`, which lets us register users online by recording their names and email addresses in a file named `reg.log`:

```
#!/usr/local/bin/perl
use CGI;
$co = new CGI;
print $co->header,
$co->start_html(-title=>'CGI Example',
-author=>'Steve',-meta=>{'keywords'=>'CGI Perl'},
-BGCOLOR=>'white',-LINK=>'red');
  if ($co->param()) {
```

```
            $! = 0;
      open FILEHANDLE, ">>reg.log";
      print FILEHANDLE "Date: " . `date`;
      print FILEHANDLE "Name: " . $co->param('name') . "\n";
      print FILEHANDLE "email: " . $co->param('email') . "\n";
      close FILEHANDLE;
            unless ($!) { print "Thanks for registering."; }
            else { print "Sorry, there was an error: $!"; }
   }
   print $co->end_html;
```

We pass two parameters to this CGI script: name and email. We do that by appending a string beginning with a question mark to the end of this script's URL, giving those parameters like this:

```
?name=Steve&email=me@here
```

Note that you can handle spaces in the parameter's values by using a plus sign (+) instead of a space. To pass this string to `reg.cgi`, we use an `HTTP GET` request, like this:

```
import java.io.*;
import java.net.*;
public class reg
{
  public static void main(String[] args)
  {
        try {
                Socket s = new Socket("localhost", 80);
                BufferedReader in = new BufferedReader(new
                InputStreamReader(s.getInputStream()));
                PrintWriter out = new PrintWriter(s.getOutputStream());
                out.print("GET " + "/yourname/cgi/reg.cgi?name
                =Steve&email=me@here\n\n");
                out.flush();
                String line;
                while ((line = in.readLine()) != null) {
                        System.out.println(line); }
        }
        catch (Exception e){ }
  }
}
```

That's all it takes. This effectively calls the `URL` `http://localhost//yourname/cgi/reg.cgi?name=Steve&email=me@here`. This is how you send data to an HTML form. When you install `reg.cgi` on the server (using your server name and the correct URL for the script) and run this program, you'll get a confirming Web page back from `reg.cgi`, which `reg.java` prints out:

```
C:\>java reg
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML>
<HEAD>
<TITLE>CGI Example</TITLE>
<LINK REV=MADE HREF="mailto:Steve">
<META NAME="keywords" CONTENT="CGI Perl">
</HEAD><BODY BGCOLOR="white" LINK="red">
Thanks for registering.
</BODY>
</HTML>
```

That's it. Now you can submit an HTML form from Java.

Handling **URL**

In Java, you handle URLs with the `URL` class—for example, you can pass a `URL` object to the `getImage()` method of the `Applet` class to load in images. Here's the inheritance diagram for the `URL` class:

```
java.lang.Object
  |____java.net.URL
```

URL, called Uniform Resource Locator, uniquely identifies or addresses information on the Internet. There are many items of software (and Java classes) that work directly with URLs. Java's URL class has several constructors (each can throw a `MalformedURLException`) and methods, which are listed in Table 20.9 and Table 20.10, respectively:

Table 20.9: Constructor of the URL class

Constructors	Does this
`URL(String spec)`	It creates a `URL` object from the `String` representation
`URL(String protocol, String host, int port, String file)`	It creates a `URL` object from the specified protocol, host, port number, and file
`URL(String protocol, String host, int port, String file, URLStreamHandler handler)`	It creates a `URL` object from the specified protocol, host, port number, file, and handler
`URL(String protocol, String host, String file)`	It creates a `URL` from the specified protocol name, host name, and file name
`URL(URL context, String spec)`	It creates a `URL` by parsing the given spec within a specified context
`URL(URL context, String spec, URLStreamHandler handler)`	It creates a `URL` by parsing the given spec with the specified handler within a specified context

Table 20.10: Methods of the URL class

Method	Does this
`boolean equals(Object obj)`	It compares two `URL`s
`String getAuthority()`	It returns the authority part of this `URL`
`Object getContent()`	It gets the contents of this `URL`
`Object getContent (Class [ ] classes)`	It returns the contents of this `URL`
`String getFile()`	It gets the file name of this `URL`
`String getHost()`	It gets the hostname of this `URL`, if applicable
`String getPath ()`	It returns the path part of this `URL`
`int getPort()`	It gets the port number of this `URL`
`String getProtocol()`	It gets the protocol name of this `URL`
`String getQuery ()`	It returns the query part of this `URL`
`String getRef()`	It gets the anchor (also known as the reference) of this `URL`
`String getUserInfo ()`	It returns the userInfo part of this `URL`
`int hashCode()`	It constructs an integer suitable for hashtable indexing
`URLConnection openConnection()`	It gets a `URLConnection` object that represents a connection to the remote object referred to by the `URL`
`InputStream openStream()`	It opens a connection to this `URL` and returns an input stream for reading from that connection
`boolean sameFile(URL other)`	It compares two `URL`s, excluding the "ref" fields
`protected void set(String protocol, String host, int port, String file, String ref)`	It sets the fields of the `URL`
`protected void set (String protocol, String host, int port, String authority, String userInfo, String path, String query, String ref)`	It sets the specified eight fields of the `URL`
`static void setURLStreamHandlerFactory (URLStreamHandlerFactory fac)`	It sets an application's URLStreamHandlerFactory. The URLStreamHandlerFactory instance constructs a stream protocol handler from a protocol name.

Table 20.10: Methods of the URL class	
Method	**Does this**
`String toExternalForm()`	It constructs a string representation of this `URL` in standard `URL` form
`String toString()`	It constructs a string representation of this `URL`
URI toURI()	It returns a `URI` equivalent to this URL

The most common form used includes the specification of the `URL` with a `String` that is similar to what you see in a browser:

```
URL(String spec)
```

The other two forms of constructors break up the `URL` into its component parts as:

```
URL(String protocol, String host, int port, String file)
URL(String protocol, String host, String file)
```

The constructors that allow you to use an existing URL as a reference context and then create a new URL from that context are:

```
URL(URL Context, String spec)
```

Here's an example that puts the URL class to work. In this case, we create a URL object for the Web page `https://www.oracle.com/java/index.html` and then use the URL class's methods to get various parts of the URL:

```
import java.net.*;
class URLDemo {
  public static void main (String args[]) throws MalformedURLException {
        URL httpurl = new URL("https://www.oracle.com/java/index.html");
        System.out.println("Protocol: "+ httpurl.getProtocol ());
        System.out.println("Port: "+ httpurl.getPort ());
        System.out.println("Host: "+ httpurl.getHost ());
        System.out.println("File: "+ httpurl.getFile ());
        System.out.println("Ext: "+ httpurl.toExternalForm ()); }
}
```

The result is shown in Figure 20.7:

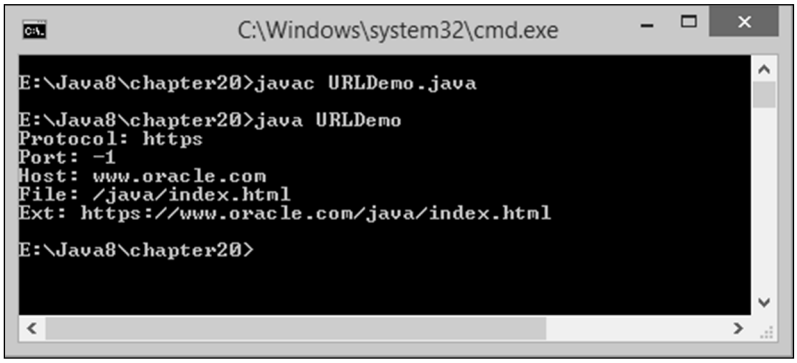


Figure 20.7: Handling URL

The result shows that the port is –1. This means that the one was not explicitly set. Once a URL object is created, the data associated with it can be easily retrieved. You can create a `URLConnection` object for accessing the actual bits or content information of a `URL`. A `URLConnection` object is created from the URL class using the `openConnection()` method. The syntax for the method is as follows:

```
url.openconnection()
openConnection() has the following general form:
URLConnection openConnection()
```

The preceding code snippet returns a `URLConnection` object related to the invocation of the URL object. It may throw an `IOException`.

Using the **URLConnection** Objects

"Hey," says the NP, "why can't I download Web pages using the URL class?" You smile and say, "For that, you can use the `URLConnection` class."

You can use the `URLConnection` class to get a great deal of information about Web pages. Here's the inheritance diagram for this class:

```
java.lang.Object
  |____java.net.URLConnection
```

You'll find the fields of the `URLConnection` class in Table 20.11, its constructor in Table 20.12, and its methods in Table 20.13:

Table 20.11: Fields of the URLConnection class

Field	Does this
`protected boolean allowUserInteraction`	It returns True if this `URL` is being examined in a context in which it can support user interactions
`protected boolean connected`	It returns False if this connection object has not created a link to the indicated URL
`protected boolean doInput`	It sets by the `setDoInput()` method
`protected boolean doOutput`	It sets by the `setDoOutput()` method
`protected long ifModifiedSince`	Some protocols skip the fetching of the object unless the object has been modified recently
`protected URL url`	It represents the remote object on the World Wide Web to which this connection is opened
`protected boolean useCaches`	It returns True if the protocol is allowed to use caching

Table 20.12: The constructor of the URLConnection class

Constructor	Does this
`protected URLConnection(URL url)`	It constructs a URL connection to the specified URL

Table 20.13: Methods of the URLConnection class

Method	Does this
`void addRequestProperty (String key, String value)`	It adds a general request property specified by a key-value pair.
`abstractvoid connect()`	It opens a communications link to the resource referenced by this URL, if such a connection has not already been established.
`boolean getAllowUserInteraction()`	It returns the value of the `allowUserInteraction` field for this object.
`int getConnectTimeout()`	It returns setting for connect timeout.
`Object getContent()`	It retrieves the contents of this `URLConnection`.
`Object getContent (Class[] classes)`	It retrieves the contents of this `URLConnection`.
`String getContentEncoding()`	It returns the value of the content-encoding header field.
`int getContentLength()`	It returns the value of the content-length header field.
`String getContentType()`	It returns the value of the content-type header field.
`long getDate()`	It returns the value of the date header field.
`boolean getDefaultUseCaches()`	It returns the default value of a `URLConnection's useCaches` flag.
`boolean getDoInput()`	It returns the value of this `URLConnection's doInput` flag.
`boolean getDoOutput()`	It returns the value of this `URLConnection's doOutput` flag.

Table 20.13: Methods of the URLConnection class

Method	Does this
`long getExpiration()`	It returns the value of the expired header field.
`static FileNameMap getFileNameMap()`	It loads filename map (a mimetable) from a data file.
`String getHeaderField(int n)`	It returns the value for the nth header field.
`String getHeaderField (String name)`	It returns the value of the named header field.
`long getHeaderFieldDate (String name,long Default)`	It returns the value of the named field parsed as date.
`int getHeaderFieldInt (String name, int Default)`	It returns the value of the named field parsed as a number.
`String getHeaderFieldKey (int n)`	It returns the key for the nth header field.
`long getIfModifiedSince()`	It returns the value of this object's `ifModifiedSince` field.
`InputStream getInputStream()`	It returns an input stream that reads from this open connection.
`long getLastModified()`	It returns the value of the last-modified header field.
`OutputStream getOutputStream()`	It returns an output stream that writes to this connection.
`Permission getPermission()`	It returns a permission object representing the permission necessary to make the connection represented by this object.
`int getReadTimeout()`	It returns setting for read timeout.
`Map<String,List<String>> getRequestProperties()`	It returns an unmodifiable `Map` of general request properties for this connection.
`String getRequestProperty (String key)`	It returns the value of the named general request property for this connection.
`URL getURL()`	It returns the value of this `URLConnection's` URL field.
`boolean getUseCaches()`	It returns the value of this `URLConnection's useCaches` field.
`static String guessContentTypeFromName (String fname)`	It tries to determine the content type of an object, based on the specified "file" component of a URL.
`static String guessContentTypeFromStream(InputS tream is)`	It tries to determine the type of an input stream based on the characters at the beginning of the input stream.
`void setAllowUserInteraction (boolean allowuserinteraction)`	It sets the value of the `allowUserInteraction` field of this `URLConnection.`
`void setConnectTimeout (int timeout)`	It sets a specified timeout value, in milliseconds, to be used when opening a communications link to the resource referenced by this `URLConnection.`
`static void setDefaultAllowUserInteraction(boo lean defaultallowuserinteraction)`	It sets the default value of the `allowUserInteraction` field for all future `URLConnection` objects to the specified value.
`static void setDefaultRequestProperty (Stringkey, String value)`	It is deprecated. The instance specific `setRequestProperty()` method should be used after an appropriate instance of `URLConnection` is obtained. Invoking this method will have no effect.
`void setDefaultUseCaches (boolean defaultusecaches)`	It sets the default value of the `UseCaches` field to the specified value.
`void setDoInput (boolean doinput)`	It sets the value of the `doInput field` for this `URLConnection` to the specified value.

Table 20.13: Methods of the URLConnection class	
Method	**Does this**
`static void setFileNameMap (FileNameMap map)`	It sets the `FileNameMap`.
`void setIfModifiedSince (long ifmodifiedsince)`	It sets the value of the `ifModifiedSince` field of this `URLConnection` to the specified value.
`void setReadTimeout (int timeout)`	It sets the read timeout to a specified timeout, in milliseconds.
`void setRequestProperty (String key, String value)`	It sets the general request property.
`void setUseCaches (boolean usecaches)`	It sets the value of the `useCaches` field of this `URLConnection` to the specified value.
`String toString()`	It returns a `String` representation of this `URLConnection`.

Here's an example in which we check on a fictitious URL: `http://www.oracle.com/technetwork/java/index.html`. Here's the code (note that we can use the `getInputStream()` method to actually get the Web page content):

```
import java.io.*;
import java.net.*;
import java.util.Date;

class URLConnectionDemo
{
  public static void main(String args[]) throws Exception
  {
        int character;
        URL url = new
        URL("http://www.oracle.com/technetwork/java/index.html");
        URLConnection urlconnection = url.openConnection();
        System.out.println("Content type: " +
        urlconnection.getContentType());
        System.out.println("Document date: " + new
        Date(urlconnection.getDate()));
        System.out.println("Last modified: " + new
        Date(urlconnection.getLastModified()));
        System.out.println("Document expires: " +
        urlconnection.getExpiration());
        int contentlength = urlconnection.getContentLength();
        System.out.println("Content length: " + contentlength);
        if (contentlength > 0)

        {
                InputStream in = urlconnection.getInputStream();
                while ((character = in.read()) != -1)
                {
                        System.out.print((char) character);
                }
                in.close();
        }
  }
}
```

The output of the code is shown in Figure 20.8, showing the URL content:

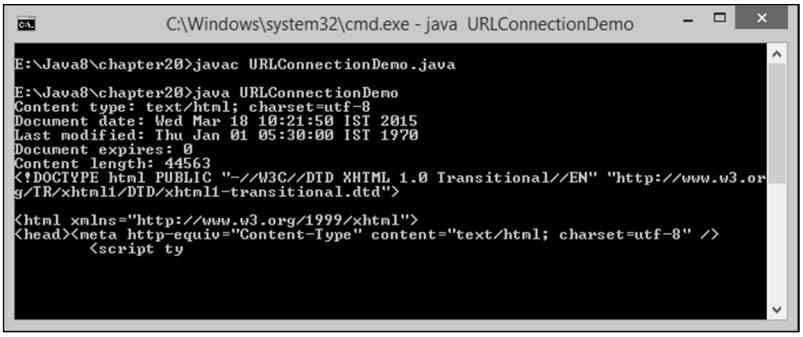


Figure 20.8: Showing the URL connection to display the information

If you use it, make sure you specify a URL in the code.

To access the attributes of a remote resource, `URLConnection` is used, which first inspects the properties of the remote object itself before transporting it locally. The HTTP protocol specification opens these attributes which are then used by the `URL` objects using the HTTP protocol.

You need to create a `URLConnection` with the help of the `openConnection()` method of a `URL` object if you want to view properties and content of the document:

```
import java.net.*;
import java.io.*;
import java.util.Date;

class URLConn
{
  public static void main (String args[]) throws Exception
  {
        int c;
        URL httpurl = new
        URL("http://www.oracle.com/technetwork/java/index.html");
        URLConnection httpurlCon = httpurl.openConnection();
        long d = httpurlCon.getDate();

        if (d==0) { System.out.println ("No date information."); }
        else
                System.out.println("Date: " + new Date(d));
                System.out.println ("Content-Type: " +
                httpurlCon.getContentType());
                d = httpurlCon.getExpiration();

        if(d == 0) { System.out.println ("No expiration information."); }
        else
                System.out.println ("Expires: " + new Date(d));
                d = httpurlCon.getLastModified();

        if (d == 0){System.out.println ("No last modified information.");}
        else
                System.out.println ("Last-Modified: " + new Date(d));
                int len = httpurlCon.getContentLength();

        if (len == -1)
                System.out.println ("Content length unavailable.");
        else { System.out.println ("Content-Length: " + len); }

        if (len != 0) {
                System.out.println ("=== Content ===");
                InputStream input = httpurlCon.getInputStream();
                int I = len;
```

```
                    while (((c = input.read()) != -1) && (--I > 0))
                    {
                            System.out.print((char) c); }
                    input.close(); }
            else { System.out.println ("No content available."); }
    }
}
```

The program establishes an HTTP connection to `www.oracle.com`, as shown in Figure 20.9:

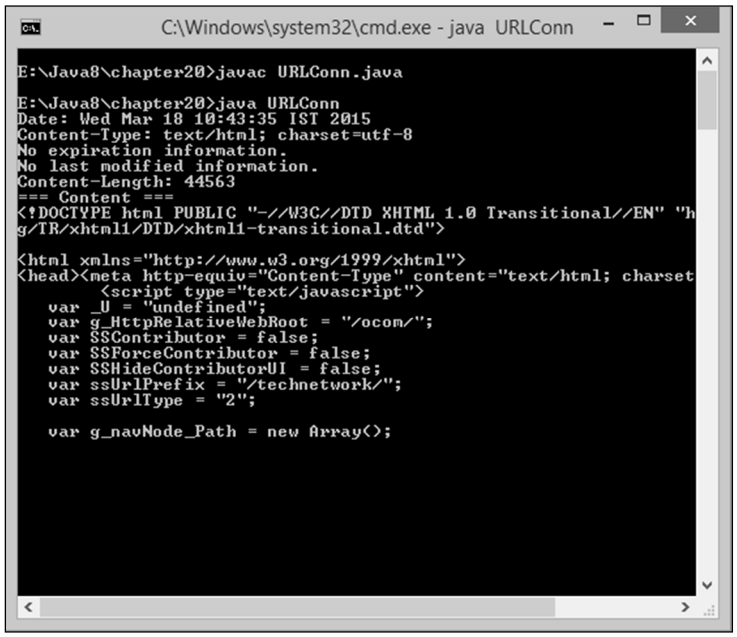


Figure 20.9: Forming a URL connection

You can list out the values of header and access the content. The output may vary depending on the conditions. The `URL` and `URLConnection` classes are required for simple programs that want to connect to HTTP servers to fetch the content.

Working with Datagrams

It is not always necessary to have a guaranteed, point-to-point channel by TCP-based protocols, because some applications might prefer to have independent packages of information which need no confirmation of its arrival or any sort of guarantee. Here comes the time of action for UDP protocol which sends small packages of data called *Datagrams* in order to transfer data from one end to the other.

The `DatagramPacket` and `DatagramSocket` classes available in the `java.net` package are used for implementing system-independent datagram communication through UDP.

In other words, Java implements `Datagrams` on top of the UDP protocol by using these two classes. The object of the `DatagramPacket` class is the data container, and the `DatagramSocket` class is used for sending or receiving the `DatagramPacket`.

DatagramPacket

The information passed through one datagram packet is independent of the other. In this, each message is routed from one machine to another based solely on information contained within that packet; different packets can be routed differently even if they are sent from a single machine regardless of the order. This means that it might arrive in any order and this delivery is not guaranteed. `DatagramPacket` defines several constructors, which you can see in Table 20.14 and the methods in Table 20.15:

Table 20.14: Constructors for the DatagramPacket class	
Constructors	**Does this**
`DatagramPacket(byte[] buf, int length)`	It constructs a `DatagramPacket` for receiving packets of length
`DatagramPacket(byte[] buf, int length, InetAddress address, int port)`	It constructs a `DatagramPacket` for sending packets of length to the specified port number on the specified host
`DatagramPacket(byte[] buf, int offset, int length)`	It constructs a `DatagramPacket` for receiving packets of length, specifying an offset into the buffer
`DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)`	It constructs a `DatagramPacket` for sending packets of length with offset to the specified port number on the specified host
`DatagramPacket(byte[] buf, int offset, int length, SocketAddress address)`	It constructs a `DatagramPacket` for sending packets of length with offset to the specified port number on the specified host
`DatagramPacket(byte[] buf, int length, SocketAddress address)`	It constructs a `DatagramPacket` for sending packets of length to the specified port number on the specified host

The constructor `DatagramPacket(byte[], int length)` specifies a buffer that is used for receiving data and the packet's size. It is used for receiving data over a datagram socket. The other syntax of constructor `DatagramPacket(byte[], int offset, int length)` also contains an offset into the buffer at which data will be stored. The third syntax of the constructor `DatagramPacket(byte[], int length, InetAddress, ipAddress, int port)` specifies a target address and a port in addition, which are used by `DatagramSocket` for determining the location to which the data in the packet will be sent.

The next constructor `DatagramPacket(byte[], int offset, int length, InetAddress, ipAddress, int port)` is used for transmitting packets starting at the particular offset into the data.

There are various methods for accessing the internal state of a `DatagramPacket` class, which are provided in Table 20.15:

Table 20.15: Methods of the DatagramPacket class	
Methods	**Does this**
`InetAddress getAddress()`	It returns the IP address of the machine to which this datagram is being sent or from which the datagram was received
`byte[] getData()`	It returns the data buffer
`int getLength()`	It returns the length of the data to be sent or the length of the data received
`int getOffset()`	It returns the offset of the data to be sent or the offset of the data received
`int getPort()`	It returns the port number on the remote host to which this datagram is being sent or from which the datagram was received
`SocketAddress getSocketAddress()`	It gets the `SocketAddress` (usually IP address + port number) of the remote host that this packet is being sent to or is coming from
`void setAddress(InetAddress iaddr)`	It sets the IP address of the machine to which this datagram is being sent
`void setData(byte[] buf)`	It sets the data buffer for this packet
`void setData(byte[] buf, int offset, int length)`	It sets the data buffer for this packet
`void setLength(int length)`	It sets the length for this packet
`void setPort(int iport)`	It sets the port number on the remote host to which this datagram is being sent
`void setSocketAddress(SocketAddress address)`	It sets the `SocketAddress` (usually IP address + port number) of the remote host to which this datagram is being sent

Datagrams Server and Client

The messages written into the window at the server and across the network to the client side, where they are shown, illustrate the concept of Datagram Server and Client.

Following is the code for server:

```
import java.net.*;
class ServerApp
{
  public static int servPort = 997;
  public static int clntPort = 998;
  public static int buff_size = 1024;

  public static  DatagramSocket ds;
  public static byte buff[] = new byte[buff_size];
  public static void Server() throws Exception
  {
        int pos =0;
        while (true)
        {
                int c = System.in.read();
                switch (c)
                {
                case -1:
                        System.out.println ("Server Quits");
                        return;
                case '\r':
                        break;
                case '\n':
                        ds.send(new DatagramPacket
                        (buff,pos,InetAddress. getLocalHost(),clntPort));
                        pos = 0;
                        break;
                default:
                        buff[pos++] = (byte) c; }
        }
  }
  public static void main (String args[]) throws Exception
  {
        ds = new DatagramSocket (servPort);
        Server();
  }
}
```

Following is the code for client:

```
import java.net.*;

class ClientApp
{
  public static int servPort = 997;
  public static int clntPort = 998;

  public static int buff_size = 1024;
  public static  DatagramSocket ds;
  public static byte buff[] = new byte[buff_size];
  public static void client() throws Exception
  {
        while(true)
        {
                DatagramPacket p = new DatagramPacket(buff, buff.length);
                ds.receive (p);
                System.out.println (new String (p.getData(), 0,
                p.getLength())); }
  }
  public static void main (String args[]) throws Exception
```

```
    {
          ds = new DatagramSocket (clntPort);
          client();
    }
}
```

The preceding programs are restricted by the `DatagramSocket` constructor to running between two ports on the local machine. To run the programs, use:

```
java ServerApp
```

in one window; this will be the server. Then run:

```
java ClientApp
```

This will be the client. Data that is typed into the window at the server side will be written across the network to the client-side, where it is displayed after a newline is received, as shown in Figure 20.10:



Figure 20.10: Connection between Client and Server

Working with **BART**

Now, let's design a client program that communicates with a server, `BartServer`. Each time a client, in our case `BartClient`, connects to the server, it asks for a host name. Then it connects to the server at that host, requests a quote, and displays the result on the screen 5 times. Now let's write the server program with the name, `BartServer2`:

```
import java.net.*;
import java.io.*;
import java.util.*;

public class BartServer2
{
  public static void main(String[] args)
  {
        int port = 1234;
        BartQuote bart = new BartQuote();
        try {
                System.out.println("BartServer 2.0");
                System.out.println("Listening on port " + port);
                ServerSocket ss = new ServerSocket(port);
                while (true) {
                        Socket s = ss.accept();
                        System.out.println("Connection established!");
                        Thread t = new Thread(new BartThread(s, bart));
                        t.start(); }
        }
        catch (Exception e) { System.out.println("System exception!"); }
  }
}
class BartThread implements Runnable
{
  private Socket s;
  private BartQuote bart;
  public BartThread(Socket socket, BartQuote bart)
  {
        this.s = socket;
        this.bart = bart; }
```

```
    public void run() {
          String client = s.getInetAddress().toString();
          System.out.println("Connected to " + client);
          try {
                Scanner in = new Scanner(s.getInputStream());
                PrintWriter out;
                out = new PrintWriter(s.getOutputStream(),
                true);
                out.println("Welcome to the Bart Server");
                out.println("Enter BYE to exit.");
                while (true) {
                      String input = in.nextLine();
                      if (input.equalsIgnoreCase("bye"))
                      break;
                      else if (input.equalsIgnoreCase("get")) {
                            out.println(bart.getQuote());
                            System.out.println("Serving " + client); }
                      else { out.println("Huh?"); }
                }
                out.println("So long, suckers!");
                s.close(); }
          catch (Exception e) { e.printStackTrace(); }
          System.out.println("Closed connection to " + client); }
}
```

Just compile the code. Now let's type the code for the client, `BartClient`, as follows:

```
import java.net.*;
import java.util.*;
import java.io.*;

public class BartClient
{
  public static void main(String[] args)
  {
        int port = 1234;
        System.out.println("Welcome to the Bart Client\n");
        Socket s = getSocket(port);
        try {
                System.out.println("Connected on port " + port);
                Scanner in =    new Scanner(s.getInputStream());
                PrintWriter out;
                out = new PrintWriter(s.getOutputStream(), true);
                in.nextLine();
                in.nextLine();
                out.println("get");
                String quote = in.nextLine();
                out.println("bye");
                s.close();
                for (int i = 0; i < 5; i++)
                      System.out.println(quote);
        }
        catch (Exception e) { e.printStackTrace(); }
  }
  private static Socket getSocket(int port)
  {
        Socket s;
        String host;
        InetAddress ip;
        Scanner sc = new Scanner(System.in);
        while (true) {
                System.out.print(
                "What server do you want to connect to?");
                host = sc.nextLine();
                try {
                      ip = InetAddress.getByName(host);
                      s = new Socket(ip, port);
```

```
                        return s; }
                catch (UnknownHostException e) {
                        System.out.println("The host is unknown."); }
                catch (IOException e) {
                        System.out.println("Network error."); }
            }
        }
    }
```

Compile and execute the preceding code. The result of the Bart program is as follows (it differs from system to system):

```
C:\> Java BartClient
Welcome to the Bart Client
What server do you want to connect to?yahoo.com
The host is unknown.
What server do you want to connect to?//appl
The host is unknown.
What server do you want to connect to?server_dt
Network error.
What server do you want to connect to?allen
Connection established!
Connected on port 1234
Connected to /192.168.0.51
Serving /192.168.0.51
Closed connection to /192.168.0.51
I will not fake seizures.
I will not fake seizures.
I will not fake seizures.
I will not fake seizures.
I will not fake seizures.
Interactive Session Ended
```

The procedure to connect to the server is placed in the `getSocket()` method. This method prompts the user for a host name. The host name is read from the console and the method returns with the user that the program can connect to.

Learning about the **java.security** Package

The `java.security` package provides classes and interfaces for security framework. The classes of this package implement access control security architecture. This implemented security architecture is easily configurable. Using this package, we can avail all support for the generation and storage of cryptographic public keys. It also supports number of cryptographic operations which includes operations for message digest and generation of signature. This package also contains classes for signed/guarded objects and random number generation. Most of the classes are provider-based. These classes define their own programming interface and applications may write to these interfaces. The implementation can be written by any third party and plugged as required. It helps developers as they can take advantage of the number of provider-based implementations and they need not rewrite the codes.

The interfaces of the `java.security` package can be found in Table 20.16 and their classes in Table 20.17:

Table 20.16: Interfaces of the java.security package

Interface	Description
`DomainCombiner`	It provides a means to dynamically update the `ProtectionDomains` associated with the current `AccessControlContext`
`Guard`	This interface represents a guard, which is used to protect access to another object
`Key`	It is a top level interface for all keys
`KeyStore.Entry`	It is a marker interface for `KeyStore` entry types
`KeyStore.LoadStoreParameter`	It is a marker interface for `KeyStore` load and store parameters
`KeyStore.ProtectionParameter`	It is a marker interface for `keystore` protection parameters

Table 20.16: Interfaces of the java.security package	
Interface	**Description**
`Policy.Parameters`	It is a marker interface for `Policy` parameters
`Principal`	This interface represents the abstract notion of a principal, which can be used to represent any entity
`PrivateKey`	It is a private key
`PrivilegedAction<T>`	It is a computation to be performed with privileges enabled
`PrivilegedExceptionAction<T>`	It is a computation to be performed with privileges enabled, that throws one or more checked exceptions
`PublicKey`	It is a public key

Table 20.17: Classes of the java.security package	
Classe	**Description**
`AccessControlContext`	It is used to make system resource access decisions based on the context it encapsulates
`AccessController`	It is used for access control operations and decisions
`AlgorithmParameterGenerator`	It is used to generate a set of parameters to be used with a certain algorithm
`AlgorithmParameterGeneratorSpi`	This class defines the Service Provider Interface (SPI) for the `AlgorithmParameterGenerator` class, which is used to generate a set of parameters to be used with a certain algorithm
`AlgorithmParameters`	It is used as an opaque representation of cryptographic parameters
`AlgorithmParametersSpi`	This class defines the Service Provider Interface (SPI) for the `AlgorithmParameters` class, which is used to manage algorithm parameters
`AllPermission`	It is a permission that implies all other permissions
`AuthProvider`	It defines `login()` and `logout()` methods for a provider
`BasicPermission`	This class extends the `Permission` class, and can be used as the base class for permissions that want to follow the same naming convention as `BasicPermission`
`CodeSigner`	It encapsulates information about a code signer
`CodeSource`	`CodeSource` class extends the concept of a codebase to encapsulate not only the location (URL) but also the certificate chains that were used to verify signed code originating from that location
`DigestInputStream`	It is a transparent stream that updates the associated message digest by using the bits going through the stream
`DigestOutputStream`	It is a transparent stream that updates the associated message digest by using the bits going through the stream
`GuardedObject`	It is an object that is used to protect access to another object
`KeyFactory`	It is used to convert keys (opaque cryptographic keys of type Key) into key specifications (transparent representations of the underlying key material), and vice versa
`KeyFactorySpi`	It defines the Service Provider Interface (SPI) for the `KeyFactory` class
`KeyPair`	It is a simple holder for a key pair (a public key and a private key)
`KeyPairGenerator`	It is used to generate pairs of public and private keys
`KeyPairGeneratorSpi`	It defines the Service Provider Interface (SPI) for the `KeyPairGenerator` class
`KeyRep`	It is standardized representation for serialized `Key` objects

Table 20.17: Classes of the java.security package

Classe	Description
`KeyStore`	It represents a storage facility for cryptographic keys and certificates.
`KeyStore.Builder`	It is a description of a to-be-instantiated KeyStore object.
`KeyStore.CallbackHandlerProtection`	It is a `ProtectionParameter` encapsulating a `CallbackHandler`
`KeyStore.PasswordProtection`	It is a password-based implementation of `ProtectionParameter`
`KeyStore.PrivateKeyEntry`	It is a `KeyStore` entry that holds a `PrivateKey` and corresponding certificate chain
`KeyStore.SecretKeyEntry`	A `KeyStore` entry that holds a `SecretKey`
`KeyStore.TrustedCertificateEntry`	A `KeyStore` entry that holds a trusted Certificate
`KeyStoreSpi`	It defines the Service Provider Interface (SPI) for the `KeyStore` class
`MessageDigest`	It provides applications, the functionality of a message digest algorithm, such as SHA-1 or SHA-256
`MessageDigestSpi`	It defines the Service Provider Interface (SPI) for the `MessageDigest` class
`Permission`	It is an abstract class for representing access to a system resource
`PermissionCollection`	It is an abstract class representing a collection of `Permission` objects
`Permissions`	It represents a heterogeneous collection of Permissions
`Policy`	This object is responsible for determining whether code executing in the Java runtime environment has permission to perform a security-sensitive operation
`PolicySpi`	It defines the SPI for the `Policy` class
`ProtectionDomain`	It encapsulates the characteristics of a domain, which encloses a set of classes whose instances are granted a set of permissions when being executed on behalf of a given set of Principals
`Provider`	It represents a "provider" for the Java Security API, where a provider implements some or all parts of Java Security
`Provider.Service`	It is description of a security service
`SecureClassLoader`	`SecureClassLoader` class extends `ClassLoader` with additional support for defining classes with an associated code source and permissions which are retrieved by the system policy by default
`SecureRandom`	It provides a cryptographically strong Random Number Generator (RNG)
`SecureRandomSpi`	It defines the SPI for the `SecureRandom` class
`Security`	It centralizes all security properties and common security methods
`SecurityPermission`	This class is for security permissions
`Signature`	This `Signature` class is used to provide applications, the functionality of a digital signature algorithm
`SignatureSpi`	It defines the SPI for the `Signature` class
`SignedObject`	It is a class for the purpose of creating authentic runtime objects whose integrity cannot be compromised without being detected
`Timestamp`	`Timestamp` class encapsulates information about a signed timestamp
`UnresolvedPermission`	This class is used to hold Permissions that were "unresolved" when the `Policy` was initialized
`URIParameter`	It is a parameter that contains a URI pointing to data intended for a `PolicySpi` or `ConfigurationSpi` implementation

The Permission Class

The two most important classes of the `java.security` package are `java.security.Permission` and `java.security.Policy`.

The `Permission` class is an abstract class in the `java.security` package which represents access to a system resource. A `permission` has a name and abstract functions defining the semantics of a particular `Permission` subclass. A Permission object may include a list of permitted actions. For example, for an object of the `java.io.FilePermission` class (which extends `Permission` class), the pathname of the file or directory is the permission name, and the actions permitted (like "read, write") on the object are represented by an action list.

All subclasses of the `Permission` class must implement the `implies()` method to compare Permissions. Here's the inheritance diagram for this class:

```
java.lang.Object
  |____java.security.Permission
```

The constructor of the `java.security.Permission` can be found in Table 20.18 and its methods can be constructor in Table 20.19:

Table 20.18: The constructor of the java.security.Permission class

Method	Does this
`Permission(String name)`	It creates a permission with a name specified

Table 20.19: Methods of the java.security.Permission class

Method	Does this
`void checkGuard (Object object)`	It implements the guard interface for a permission
`abstract Boolean equals (Object obj)`	It checks two `Permission` objects for equality
`abstract String getActions()`	It gives actions as a `String`
`String getName()`	It gives name of this `Permission`
`abstract int hashCode()`	It gives hash code value for this `Permission` object
`abstract Boolean implies (Permission permission)`	It checks if the specified permission's actions are "implied by" this object's actions
`PermissionCollection newPermissionCollection()`	It yields an empty `PermissionCollection` for a given Permission object, or null if one is not defined
`String toString()`	It yields a string describing this `Permission`

The Policy Class

An object of the `Policy` class is used to find if the code executing in Java runtime environment has permission to perform security-sensitive operations.

In the runtime, there is only one `Policy` object installed at a time. A `Policy` object is installed by using the `setPolicy()` method and can be obtained by using the `getPolicy()` method. If no `Policy` object is installed, `getPolicy()` installs a default `Policy` implementation. To provide a custom implementation, we can subclass the `Policy` class. After an instance of the `Policy` object is installed, Java runtime calls the `implies()` method to find whether the running code can perform operations protected by SecurityManager.

The `refresh()` method enables the `Policy` object in refreshing/reloading its data. Here's the inheritance diagram for this class:

```
java.lang.Object
  |____java.security.Policy
```

The nested classes of the `java.security.Policy` class are given in Table 20.20, the fields in Table 20.21, and the methods in Table 20.22:

Table 20.20: Nested class of the java.security.Policy class

Class	Description
`static interface Policy.Parameters`	It represents a marker interface for Policy parameters

Table 20.21: Field summary of the java.security.Policy class

Field	Description
`static PermissionCollection UNSUPPORTED_EMPTY_COLLECTION`	It is a read-only empty `PermissionCollection` instance

Table 20.22: Methods of the java.security.Policy class

Method	Does this
`static Policy getInstance(String type, Policy.Parameters params)`	It yields a `Policy` object of the specified type
`static Policy getInstance(String type, Policy.Parameters params, Provider provider)`	It yields a `Policy` object of the specified type
`static Policy getInstance(String type, Policy.Parameters params, String provider)`	It yields a `Policy` object of the specified type
`Policy.Parameters getParameters()`	It yields `Policy` parameters
`PermissionCollection getPermissions(CodeSource codesource)`	It yields a `PermissionCollection` object containing the set of permissions granted to the specified `CodeSource`
`PermissionCollection getPermissions(ProtectionDomain domain)`	It yields a `PermissionCollection` object containing the set of permissions granted to the specified `ProtectionDomain`
`static Policy getPolicy()`	It yields the installed `Policy` object
`Provider getProvider()`	It yields the Provider of this `Policy`
`String getType()`	It yields the type of this `Policy`
`boolean implies(ProtectionDomain domain, Permission permission)`	It evaluates the global policy for the permissions granted to the `ProtectionDomain` and tests whether the permission is granted
`void refresh()`	It refreshes/reloads the policy configuration
`static void setPolicy(Policy p)`	It sets the system-wide `Policy` object

Let us see the implementation of the Policy class in the following solution:

```
import java.security.*;
import java.util.Enumeration;

public class SecurityApp {
  public static void main(String args[]) {
    ProtectionDomain domain1 = "".getClass().getProtectionDomain();

    PermissionCollection pc = Policy.getPolicy().getPermissions(domain1);

    Enumeration e1 = pc.elements();
    for (; e1.hasMoreElements();) {
      Permission per = (Permission) e1.nextElement();
```

```
            System.out.println(per);

        }
      }
    }
```

The output of the preceding code is shown in Figure 20.11:

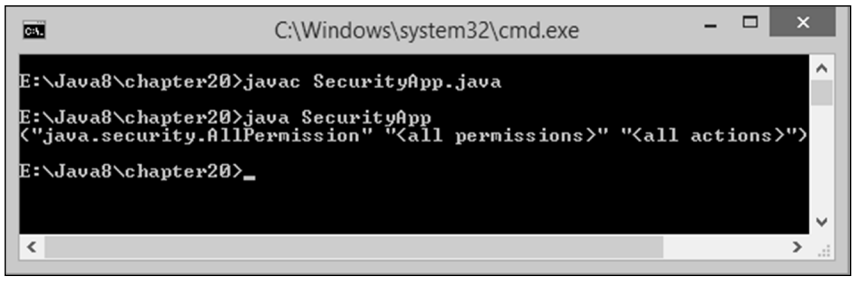


Figure 20.11: Showing the permissions for the user

Summary

In this chapter, you have learned about networking interfaces and classes, the `InetAddresses`, and caching `InetAddresses`. Then, we have explained how to create and use sockets, create TCP clients and servers. Then, we have discussed the **whois** example, as well as how to submit an HTML form. Next, you have learned to use the URL, URL connection objects. You have also come to know datagram packets, datagram server and client. Toward the end, you have learned BART and `java.security` package.

In the next chapter, you will learn about the Collections framework.



21

Collections

If you need an immediate solution to:	*See page:*
Using the Collection Interface	776
The Queue Interface	777
The List Interface	777
The Set Interface	778
The SortedSet Interface	779
Using the Collection Classes	779
Using the Comparator Interface	792
Using the Iterator Interface	793
Using the ListIterator Interface	793
Using the AbstractMap Class	795
Using the HashMap Class	796
Using the TreeMap Class	799
Using the Arrays Class	801
Learning the Fundamentals of Enumerations	804
The Legacy Classes and Interfaces	810
Using the Aggregate Operations	818
Using the java.util.function Package	819

In Depth

In this chapter, we'll take a look at the collection classes in Java, which are supported by `java.util`. As you can infer from the name, the collection classes let you group elements in various ways. They collection classes also define several methods that provide easier way of working with items. These classes are important, not just for their utility but because many other Java methods use or return objects of these classes, such as the `ArrayList` and `HashMap` classes.

The original Java release did not introduce Collections within it. So Java used ad hoc classes, such as Dictionary, Vector, Stack, and Properties. The drawback was that these classes lacked a unifying scheme and were not designed to be easily extended or adapted. Collections were finally added by J2SE 1.2. Java SE 8 has significantly increased the power and streamlined the use of collections framework.

The Collection Interfaces

Here are six collection interfaces:

- **Collection**—It is the top of the collection hierarchy. It supports basic grouping of elements.
- **Deque**—It extends `Queue` to handle a double-ended queue. (Added in Java SE 6).
- **List**—It extends `Collection` to implement lists of objects.
- **Set**—It extends `Collection` to implement sets, in which all elements must be unique.
- **Map**—It does not have duplicate keys and each key should map to at least one value. This interface generally maps values with keys.
- **SortedSet**—It extends `Set` to implement a sorted set.
- **SortedMap**— It extends `Map` to implement a sorted map.
- **Queue**—It extends `Collection` to handle special type of lists.

The implementations of the interfaces are provided in Table 21.1:

Table 21.1: Implementations of the Interfaces

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List
`Collection`	`HashSet`	`ArrayList`	`TreeSet`	`LinkedList`
`Set`	`HashSet`		`TreeSet`	
`SortedSet`			`TreeSet`	
`List`		`ArrayList`		`LinkedList`
`Map`	`HashMap`		`TreeMap`	
`SortedMap`			`TreeMap`	
`Queue`				`LinkedList`
`Deque`				`LinkedList`

You'll see the details in this chapter; in particular, there's a standard set of classes that implement the collection interfaces—the collection classes.

The Collection Classes

Here are the standard collection classes that implement the collection interfaces:

- `AbstractCollection`—It implements the `Collection` interface.
- `AbstractList`—It extends `AbstractCollection` and implements the `List` interface.
- `AbstractQueue`—It extends `AbstractCollection` and implements parts of the `Queue` interface.
- `AbstractSequentialList`—It extends `AbstractList` into a sequential (not random access) list.

- `LinkedList`—It extends `AbstractSequentialList` into a linked list, where each element knows where the next element is.
- `ArrayList`—It implements a dynamic (resizable) array.
- `ArrayDeque`—It implements a dynamic double-ended queue by extending `AbstractCollection` and implementing the `Deque` interface (added by Java SE 6).
- `AbstractSet`—It extends `AbstractCollection` and implements the `Set` interface.
- `HashSet`—It extends `AbstractSet` to be used with a hash.
- `TreeSet`—It extends `AbstractSet` to implement a set stored in tree form.

Note the `HashSet` class, which implements a set using a hash internally. In Java, *hash* stores information using hashing; this converts a key to a *hashcode* that's then used to access the corresponding value. You don't deal with the hashcode yourself—it's generated internally in Java.

The **Map** Interfaces

You can also use maps in Java. A *map* stores data in key/value pairs, much like an array, where the indexes are, themselves, objects. Typically, keys are strings, and you can look up an object in a map by using that object. Here are the map interfaces:

- `Map`—It implements a map.
- `Map.Entry`—The inner class of `Map` that describes a key/value pair in a map.
- `NavigableMap`— It extends `SortedMap` to handle the retrieval of entries based on closest match searches (added by Java SE 6).
- `SortedMap`—It extends `Map` to keep the keys in ascending order.

Java also derives some standard classes from these interfaces—the map classes.

The **Map** Classes

Here are the standard map classes defined by Java:

- `AbstractMap`—It implements the `Map` interface.
- `HashMap`—It extends `AbstractMap` using a hash.
- `TreeMap`—It extends `AbstractMap` using a tree.
- `WeakHashMap`—It extends `AbstractMap` using a hash with weak keys, which means that elements whose keys are no longer in use will be discarded.

You'll also see other classes here that are not technically members of the collection framework—`Arrays`, `Vector`, `Stack`, `Hashtable`, and others. These older collections have been rebuilt on collection framework functionality. In fact, most standard computing collections are implemented in Java in various ways; for example, the `TreeSet` class implements a set using a tree internally, which means access time is quick. `HashSet` implements a set using a hash internally, which means that adding and removing elements from such a set should always take the same amount of time, no matter how large the set grows.

Collections Framework Enhancements in Java SE 8

The Java platform provides support for the collection framework. It is known that a collection is an object representing the group of objects. The collection framework is an architecture representing and manipulating collections.

The collections framework enhancements in Java SE 8 are as follows:

- Support for lambda expressions, streams, and aggregate operations
- Performance improvement for HashMaps with key collisions
- Improved type inference

Let's learn more about these enhancements in detail in the following subsections.

Support for Lambda Expressions, Streams, and Aggregate Operations

In Java SE8, the Java Collections Framework supports lambda expressions, streams, and aggregate operations. Moreover, the Stream Application Program Interface (API) is integrated into the Collections API, and as a result, bulk operations on collections, such as sequential or parallel map-reduce transformations, can be performed. Let's now first learn about the enhancements in lambda expression and streams in Java SE 8.

Lambda Expressions and Streams in Java SE 8

In Java SE 8, new classes are added and improvement in the existing classes has been made for taking advantage of lambda expressions and streams. The new and enhanced classes are available in the following packages:

- `java.util`—Integrates the Java Collections Framework with streams and provides general utility functionality used by streams
- `java.util.function`—Includes general purpose functional interfaces that provide target types for lambda expressions and references for method
- `java.util.stream`—Includes the majority of interfaces and classes that provide functionality to streams and aggregate operations

Besides the additions made to the existing classes, some other additions include methods that accept instances of functional interfaces, which can be invoked with lambda expressions or method references. The `java.util.function` and `java.util.stream` packages are two new packages included in Java SE8. Let's discuss about each in detail.

The java.util.function Package

The `java.util.function` package includes interfaces that can be used by the JDK and with the code developed by a user. The package also includes enough functionality that can meets the common requirements of lambda expressions. This package also provides functional interfaces such as FileFilter that fulfills specific purposes.

The interfaces in the `java.util.function` package are annotated with FunctionalInterface. However, putting this annotation is not compulsory for making the compiler to identify an interface as a functional interface. The annotations are merely used for capturing design intent and providing help to the compiler to determine the accidental violations of design intent.

Functional interfaces generally focus on abstract concepts such as functions, actions, or predicates. While using functional interfaces or referring the variables typed as functional interfaces, you need to refer to the abstract concepts; for example, you can use "this function" in place of "the function represented by this object." The different types of functional interfaces and their description are presented in Table 21.2:

Table 21.2: Types of Interfaces and their Description

Interface	Description
`BiConsumer<T,U>`	Signifies an operation that takes two input arguments and gives no result
`BiFunction<T,U,R>`	Signifies a function that takes two arguments and generates a result
`BinaryOperator<T>`	Signifies an operation on two operands of the same type, providing a result of the same type as the operands
`BiPredicate<T,U>`	Signifies a predicate (boolean-valued function) of two arguments
`BooleanSupplier`	Signifies a supplier of boolean-valued results
`Consumer<T>`	Signifies an operation that takes a single input argument and gives no result
`DoubleBinaryOperator`	Signifies an operation on two double-valued operands and generating a double-valued result
`DoubleConsumer`	Signifies an operation that gets a single double-valued argument and gives no result
`DoubleFunction<R>`	Signifies a function that takes a double-valued argument and generates a result
`DoublePredicate`	Signifies a predicate (boolean-valued function) of one double-valued argument
`DoubleSupplier`	Signifies a supplier of double-valued results

Table 21.2: Types of Interfaces and their Description

Interface	Description
`DoubleToIntFunction`	Signifies a function that takes a double-valued argument and generates an int-valued result
`DoubleToLongFunction`	Signifies a function that takes a double-valued argument and generates a long-valued result
`DoubleUnaryOperator`	Signifies an operation on a single double-valued operand that generates a double-valued result
`Function<T,R>`	Signifies a function that takes one argument and generates a result
`IntBinaryOperator`	Signifies an operation on two int-valued operands and generating an int-valued result
`IntConsumer`	Signifies an operation that takes a single int-valued argument and gives no result
`IntFunction<R>`	Signifies a function that takes an int-valued argument and generates a result
`IntPredicate`	Signifies a predicate (boolean-valued function) of one int-valued argument
`IntSupplier`	Signifies a supplier of int-valued results
`IntToDoubleFunction`	Signifies a function that takes an int-valued argument and generates a double-valued result
`IntToLongFunction`	Signifies a function that takes an int-valued argument and generates a long-valued result
`IntUnaryOperator`	Signifies an operation on a single int-valued operand that generates an int-valued result
`LongBinaryOperator`	Signifies an operation on two long-valued operands and generates a long-valued result
`LongConsumer`	Signifies an operation that takes a single long-valued argument and gives no result
`LongFunction<R>`	Signifies a function that takes a long-valued argument and generates a result
`LongPredicate`	Signifies a predicate (boolean-valued function) of one long-valued argument
`LongSupplier`	Signifies a supplier of long-valued results
`LongToDoubleFunction`	Signifies a function that takes a long-valued argument and generates a double-valued result
`LongToIntFunction`	Signifies a function that takes a long-valued argument and generates an int-valued result
`LongUnaryOperator`	Signifies an operation on a single long-valued operand that generates a long-valued result
`ObjDoubleConsumer<T>`	Signifies an operation that takes an object-valued argument and a double-valued argument and gives no result
`ObjIntConsumer<T>`	Signifies an operation that takes an object-valued argument and an int-valued argument and gives no result
`ObjLongConsumer<T>`	Signifies an operation that takes an object-valued argument and a long-valued argument and gives no result
`Predicate<T>`	Signifies a predicate (boolean-valued function) of one argument
`Supplier<T>`	Signifies a supplier of results
`ToDoubleBiFunction<T,U>`	Signifies a function that takes two arguments and generates a double-valued result
`ToDoubleFunction<T>`	Signifies a function that generates a double-valued result
`ToIntBiFunction<T,U>`	Signifies a function that takes two arguments and generates an int-valued result
`ToIntFunction<T>`	Signifies a function that generates an int-valued result
`ToLongBiFunction<T,U>`	Signifies a function that takes two arguments and generates a long-valued result
`ToLongFunction<T>`	Signifies a function that generates a long-valued result
`UnaryOperator<T>`	Signifies an operation on a single operand that produces a result of the same type as its operand

The java.util.stream Package

The stream class is the main abstraction provided in this package. Classes such as Stream, IntStream, LongStream, and DoubleStream are streams over objects and the primitive int, long, and double types. Streams can be obtained by several ways, which are as follows:

- They can be obtained from a `Collection` using the `stream()` and `parallelStream()` methods.
- They can be obtained from an array using the `Arrays.stream (Object[])` method.
- They can be obtained from static factory methods on the stream classes, such as `Stream.of(Object[])`, `IntStream.range(int,int)`, or `Stream.iterate(Object, UnaryOperator)`.
- The lines of a file can be obtained from the `BufferedReader.lines()` method.
- Streams of file paths can be obtained from the methods in `Files`.
- Streams of random numbers can be obtained from the `Random.ints ()` method.
- Various other stream-bearing methods in the JDK are `BitSet.stream ()`, `Pattern.splitAsStream (java.lang.CharSequence)`, and `JarFile.stream()`.

The various interfaces available in the `java.util.stream` package and their description are listed in Table 21.3:

Table 21.3: Interfaces and their Description

Interface	Description
`BaseStream<T,S extends BaseStream<T,S>>`	Refers to the base interface for streams, which are sequences of elements that support sequential and parallel aggregate operations
`Collector<T,A,R>`	Refers to a mutable reduction operation that accumulates input elements into a mutable result container, optionally transforming the accumulated result into a final representation after the processing of all the input elements
`DoubleStream`	Refers to a sequence of primitive double- valued elements which support sequential and parallel aggregate operations
`DoubleStream.Builder`	Refers to a mutable builder for a `DoubleStream`
`IntStream`	Refers to a sequence of primitive int-valued elements which support sequential and parallel aggregate operations
`IntStream.Builder`	Refers to a mutable builder for an `IntStream`
`LongStream`	Refers to a sequence of primitive long-valued elements which support sequential and parallel aggregate operations
`LongStream.Builder`	Refers to a mutable builder for a `LongStream`
`Stream<T>`	Refers to a sequence of elements which support sequential and parallel aggregate operations
`Stream.Builder<T>`	Refers to a mutable builder for a `Stream`

Table 21.4 presents the classes in the `java.util.stream` package:

Table 21.4: Classes in the java.util.stream Package and their Description

Class	Description
`Collectors`	Contains implementations of `Collector` that implements several useful reduction operations, such as gathering elements into collections, summarizing elements on the basis of different criteria, etc.
`StreamSupport`	Provides low-level utility methods to create and manipulate streams

The main differences between stream and collections are as follows:

- **No storage—**A stream does not store elements as it is not a data structure, whereas a collection does.
- **Functional nature—**Whenever an operation is performed on a stream, it produces a result and therefore does not alter its source. For example, when the filtering operation is performed on a stream, which is retrieved from a collection, it generates a new `Stream`. This new stream does not contain filtered elements and is created instead of deleting elements present in the source collection. On the other hand, in a collection, the elements are deleted from the existing collection.
- **Laziness-seeking—**Several operations on stream can be lazily implemented due to which optimization can take place. These operations may include filtering, mapping, or duplicate removal. Generally, stream operations are broken down into intermediate (or `Stream`-producing) operations and terminal (or value- or side-effect-producing) operations. Intermediate operations are always considered lazy. On the other hand, optimization cannot take place in a collection.
- **Possibly unbounded—**Streams do not have a finite size unlike collections.
- **Consumable—**The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source. However, in a collection, the elements can be navigated again and again.

Aggregate Operations

An operation that is performed on a data structure rather than on an individual element is known as an aggregate operation. Aggregate operations, like forEach, give an impression that these are like iterators. However, they are different from iterators in the following ways:

- **They use internal iteration—**Aggregate operations perform internal iteration as they do not contain a method like next, which enables them to iterate to the next element of the collection. With the help of internal delegation, your application identifies the collection over which it iterates. But the JDK identifies the iteration over the collection. On the other hand, with the help of external iteration, your application can determine about the collection over which it iterates and the way it iterates. However, there is a limitation with external iterators as they can only iterate sequentially over the elements of a collection. This limitation does not exist with internal iterators. Moreover, the internal iteration takes benefits of parallel computing, which involves breaking a problem into subproblems and then solving them simultaneously.
- **They process elements from a stream—**Aggregate operations do not process elements directly from a collection, but directly from a stream. Therefore, they are also called stream operations.
- **They support behavior as parameters—**The lambda expressions can also be specified as parameters for most of the aggregate operations which allow customization of the behavior of a specific aggregate operation.

To better understand the concept of aggregate operations, consider a scenario in which you are developing a social networking site. You want to provide the facility that allows an administrator to perform different actions, such as sending a message to the members of the social networking application, which satisfy certain criteria. Suppose the members of this social networking application are represented by the following SNMember class:

```
public class SNMember {
    public enum Gender {
        M, FM
    }
    String sname;
    LocalDate mem_birthday;
    Gender sex;
    String email;
    // ...
    public int getMemage() {
        // ...
    }
    public String getMemname() {
        // ...
    }
}
```

Consider the following code snippet which prints the name of all members contained in the collection memlist using the for-each loop:

```
for (SNMember sn : memlist) {
    System.out.println(sn.getMemname());
}
```

Now, consider the following code snippet which prints all the members present in the collection memlist, but with the use of aggregate operation forEach:

```
memlist
    .stream()
    .forEach(e -> System.out.println(e.getMemname());
```

In the previous code snippet, the aggregate operation for-each loop is used.

Pipelines and Streams

A pipeline refers to a sequence of aggregate operations. Consider the following code snippet that prints the male members available in the collection memlist with a pipeline that comprises the aggregate operations such as filter and forEach:

```
 memlist
.stream()
.filter(e -> e.getsex() == SNMember.Gender.M)
.forEach(e -> System.out.println(e.getMemname()));
```

A pipeline consists of the following components:

- **Source**—It might be a collection, an array, a generator function, or an I/O channel. In the previous code snippet, the example of a source is the collection memlist.
- **Zero or more intermediate operations**— An intermediate operation, such as filter, produces a new stream.
- **Stream**—A stream does not store elements; instead, it carries values from a source through a pipeline. In the previous code snippet, a stream is created from the collection memlist by invoking the method stream.
- **Predicate**—A new stream is returned by the filter operation. This stream comprises those elements that match its predicate. In the preceding code snippet, the predicate is the lambda expression e -> e.getsex() == SNMember.Gender.M, which gives the boolean value(true or false). It returns true if the sex field of object e has the value SNMember.Gender.M. Subsequently, the filter operation returns a stream that comprises all male members in the collection memlist.
- **Terminal operation**—A pipleline also comprises a terminal operation, such as forEach, which generates a non-stream result, such as a primitive value, a collection, or no value at all. In the preceding code snippet, the parameter of the forEach operation is the lambda expression e -> System.out.println(e.getMemname()), which invokes the method getMemname on the object e.

Performance Improvement for HashMaps with Key Collisions

The performance has been improved for HashMap objects where lots of collisions occur in the keys. A String hash function included in Java 7 update 6 is now removed from JDK 8 along with the jdk.map.althashing.threshold system property. Hash bins, having very large number of colliding keys, enhance performance by saving their entries in a balanced tree and not in the linked list. This modification in JDK 8 implies only to HashMap, LinkedHashMap, and ConcurrentHashMap.

This modification might bring a change to the order of iteration in HashMap and HashSet in very rare situations. Any specific order of iteration is not mentioned for HashMap objects and any code that relies on iteration order should be fixed.

There is no change in the `java.util.Hashtable` class other than obsoleting the feature introduced in 7u6. The features added in Java 7 update 6 only apply to WeakHashMap and Hashtable, but these have been removed in JDK 8.

Improved Type Inference

The Java compiler takes the benefit of target typing for inferring the type parameters belonging to a generic method invocation. The *target type* of an expression refers to the data type that is expected by the Java compiler

depending on the location of appearance of expression. For example, earlier, that is, in Java SE7, an assignment statement's target type is used for type inference. However, in case of Java SE 8, the target type for type inference is used. Consider the following code snippet in which a method invocation's target types are used for inferring the data types of its arguments:

```
List<String> strLst = new ArrayList<>();
strLst.add("B");
strLst.addAll(Arrays.asList());
```

In the preceding code snippet, the method `addAll` is expecting a `Collection` object as its parameter and the method `Arrays.asList` returns a `List` object. This will work as `List` in a subtype of `Collection`.

Now, consider the case of generics; the target type of `addAll` is `Collection<? extends String>`. On the other hand, the `Arrays.asList()` method returns a `List<T>` instance. In Java SE 8, the compiler infers that the value of the type variable `T` is `String`. The compiler can infer this by using the target type `Collection<? extends String>`.

That's it for the overview of what's in this chapter. There's a lot coming up next— the collection framework is very large. It's time to turn to the "Immediate Solutions" section.

Immediate Solutions

Using the **Collection** Interface

The `Collection` interface revolves around the objects to provide maximum generality. For example, all general-purpose collection executions have a constructor that takes Collection parameter. This constructor, known as a conversion constructor, initializes the new collection to hold all of the elements in the intended location, whatever be the collection's subinterface or implementation type. You can say that it permits you to convert the collection's type.

Consider an example that you have a `Collection<String> co`, which can be a List, a Set, or another kind of Collection. It creates a new `ArrayList` (an implementation of the `List` interface), which consists of all the elements in co. The code using the `Collection<String> co` is as follows:

```
    List<String> list = new ArrayList<String>(co);
```

The foundation of the collection framework is the `Collection` interface, and because the collection classes implement this interface, we'll take a look at its methods here for reference. You'll find the methods for the `Collection` interface in Table 21.5:

Table 21.5: Methods of the Collection interface

Method	Does this
`boolean add(E e)`	It adds the given element.
`boolean addAll (Collection c)`	It adds all the elements in the given collection.
`void clear()`	It removes all the elements from this collection.
`boolean contains(E e)`	It returns True if this collection contains the given element.
`boolean containsAll(Collection c)`	It returns True if this collection contains all the elements in the given collection.
`boolean equals(E e)`	It compares the given object with this collection for equality.
`int hashCode()`	It gets the hashcode value for this collection.
`boolean isEmpty()`	It returns True if this collection has no elements.
`Iterator iterator()`	It gets an iterator over the elements in this collection.
`boolean remove(E e)`	It removes a single instance of the given element.
`boolean removeAll(Collection c)`	It removes all of this collection's elements that are also contained in the given collection (Optional Operation).
`default boolean` removeIf(Predicate<? super E> filter)	It removes all the elements of this collection which satisfy the given predicate.
`boolean retainAll(Collection c)`	It keeps only the elements in this collection which are contained in the given collection (Optional Operation).
`int size()`	It gets the number of elements in this collection.
`default` Spliterator<E> HYPERLINK "http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html" \l "spliterator--" spliterator`()`	It creates a Spliterator over the elements in this collection.
`default` Stream<E> stream`()`	It returns a sequential Stream with this collection as its source.
`Object[] toArray()`	It gets an array containing all the elements in this collection.
`<T> T[] toArray(T[ ] a)`	It gets an array containing all the elements in this collection whose type is that of the given array.

The **Queue** Interface

A Queue is a collection of elements to hold them before processing starts. Besides basic Collection operations, queues provide additional insertion, deletion, and inspection operations. The `Queue` interface follows:

```
public interface Queue<E> extends Collection<E>
 {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

Each Queue method exists in the following two forms:

1. It throws an exception if the operation fails.
2. It returns a special value if the operation fails (either null or false, depending on the operation).

The methods of the `Queue` interface are listed in Table 21.6:

Table 21.6: Methods of the Queue interface

Method	Does this
`boolean add(E e)`	It inserts specified elements into this queue and returns true. If no available space, then it throws an `IllegalStateExcetion`.
`E element()`	It retrieves and returns the element at the head of this queue.
`boolean offer(E e)`	Ii inserts element and returns true if the element is inserted into the queue.
`E peek ()`	It returns the item at the head of this queue without removing it from the queue. It throws an exception if the queue is empty and returns null.
`E pool()`	It retrieves and removes the head of this queue. If the queue is empty, then it returns null.
`E remove()`	It retrieves and removes the head of this queue.

The **List** Interface

The `List` interface is the foundation of classes such as `LinkedList` and `ArrayList`. It is basically an extension of the `Collection` interface that stores the behavior of a collection. So, we'll take a look at the methods of this interface for reference. The methods of the `List` interface are provided in Table 21.7:

Table 21.7: Methods of the List interface

Method	Does this
`void add (int index, E element)`	It inserts the given element at the given position in this list.
`boolean add(E e)`	It adds the given element to the end of this list.
`boolean addAll(Collection<? extends E> c)`	It adds all the elements in the given collection to the end of this list.
`boolean addAll(int index, Collection<? extends E> c)`	It inserts all the elements in the given collection into this list.
`void clear()`	It removes all the elements from this list.
`boolean contains(Object o)`	It returns True if this list contains the given element.
`boolean containsAll (Collection<?> c)`	It returns True if this list contains all the elements of the given collection.
`boolean equals(Object o)`	It compares the given object with this list for equality.
`Object get(int index)`	It gets the element at the given position in this list.

Table 21.7: Methods of the List interface	
Method	**Does this**
`int hashCode()`	It gets the hashcode value for this list.
`int indexOf(Object o)`	It gets the index of the first occurrence of the given element in this list or -1 if this list does not contain the element.
`boolean isEmpty()`	It returns True if this list contains no elements.
`Iterator iterator()`	It gets an iterator over the elements in this list in proper sequence.
`int lastIndexOf(Object o)`	It gets the index of the last occurrence of the given element in this list or -1 if this list does not contain the element.
`ListIterator listIterator()`	It gets a list iterator over the elements in this list (in proper sequence).
`ListIterator listIterator (int index)`	It gets a list iterator of the elements in this list, starting at the given position in this list.
`Object remove(int index)`	It removes the element at the given position in this list.
`boolean remove(Object o)`	It removes the first occurrence in this list of the given element.
`boolean removeAll (Collection<?> c)`	It removes from this list all of its elements that are contained in the given collection (optional operation).
`default replaceAll(UnaryOperator<E> operator)`	It replaces each element of this list with the result of applying the operator to that element.
`boolean retainAll (Collection<?> c)`	It keeps only the elements in this list that are contained in the given collection.
`E set(int index, E element)`	It replaces the element at the given position in the list with the new element.
`int size()`	It gets the number of elements in this list.
`default void sort(Comparator<? super E> c)`	It sorts this list according to the order induced by the specified Comparator.
`default Spliterator<E> spliterator()`	It creates a Spliterator over the elements in this list.
`List<E> subList(int fromIndex, int toIndex)`	It returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
`List subList(int fromIndex, int toIndex)`	It gets a view of the section between the given `fromIndex` (inclusive) and `toIndex`.
`Object[] toArray()`	It gets an array containing all the elements in this list in proper sequence (from first to last element).
`<T> T[] toArray (T[ ] a)`	It gets an array containing all the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

The **Set** Interface

The `Set` interface is the foundation of classes such as `HashSet` and `TreeSet`, and it is used by the collection framework to implement sets. Therefore, we'll take a look at this interface for reference. The methods of the `Set` interface are provided in Table 21.8:

Table 21.8: Methods of the Set Interface	
Method	**Does this**
`boolean add(E e)`	It adds the given element to this set.
`boolean addAll (Collection<? extends E> c)`	It adds all the elements in the given collection.
`void clear()`	It removes all the elements from this set.

Table 21.8: Methods of the Set Interface	
Method	**Does this**
`boolean contains(Object o)`	It returns True if this set contains the given element.
`boolean containsAll (Collection<?> c)`	It returns True if this set contains all the elements of the given collection.
`boolean equals(Object o)`	It compares the given object with this set for equality.
`int hashCode()`	It gets the hashcode value for this set.
`boolean isEmpty()`	It returns True if this set contains no elements.
`Iterator iterator()`	It gets an iterator over the elements in this set.
`boolean remove(Object o)`	It removes the given element from this set if it is present.
`boolean removeAll (Collection<?> c)`	It removes all elements contained in the given collection.
`boolean retainAll (Collection<?> c)`	It keeps only the elements in this set that are contained in the given collection.
`int size()`	It gets the number of elements in this set (its cardinality).
`Object[] toArray()`	It gets an array containing all the elements in this set.
`<T> T[] toArray (T[ ] a)`	It gets an array containing all the elements in this set whose runtime type is that of the given array.

The **SortedSet** Interface

The `SortedSet` interface maintains a sorted set, and you'll find the methods of this interface in Table 21.9:

Table 21.9: Methods of the SortedSet Interface	
Method	**Does this**
`Comparator<? Super E> comparator()`	It returns the comparator used to order the elements in this set or null if this set uses the natural ordering.
`E first()`	It gets the first (lowest) element currently in this set.
`SortedSet headSet (Object toElement)`	It gets a view of the section of this set whose elements are strictly less than `toElement`.
`E last()`	It gets the last (highest) element currently in this set.
`default Spliterator<E> spliterator()`	It creates a Spliterator over the elements in this sorted set.
`SortedSet<E> subSet (E fromElement, E toElement)`	It gets a view of the portion of this set whose elements range from `fromElement` to `toElemen`.
`SortedSet<E> tailSet (E fromElement)`	It gets a view of the portion of this set whose elements are greater than or equal to `fromElement`.

Using the **Collection** Classes

We have discussed earlier that a collection is a series of items of the same type unlike arrays. The main drawback of arrays was that you could not know or predict the number of items of the list. The solution to this was to create your own list. The `Collection` class contains exclusive static methods that operate on or return collections. It contains polymorphic (more than one feature) algorithms that operate on collections, "wrappers", which return a new collection.

*The **AbstractCollection** Class*

The `AbstractCollection` class is the implementation of the `Collection` interface upon which many Java collection classes are built, so we'll take a look at `AbstractCollection` for reference. Here's the inheritance diagram for this class:

```
java.lang.Object
  |____java.util.AbstractCollection
```

You'll find the constructor for the `AbstractCollection` class in Table 21.10 and its methods in Table 21.11:

Table 21.10: The constructor of the AbstractCollection class

Constructor	Does this
`protected AbstractCollection()`	It constructs an `AbstractCollection` object.

Table 21.11: Methods of the AbstractCollection class

Method	Does this
`boolean add(E e)`	It ensures that this collection contains the given element.
`boolean addAll(Collection<? extends E> c)`	It adds all the elements in the given collection to this collection.
`void clear()`	It removes all the elements from this collection.
`boolean contains(Object o)`	It returns True if this collection contains the given element.
`boolean containsAll (Collection<?> c)`	It returns True if this collection contains all the elements in the given collection.
`boolean isEmpty()`	It returns True if this collection contains no elements.
`abstract Iterator iterator()`	It gets an iterator over the elements contained in this collection.
`boolean remove(Object o)`	It removes the given element from this collection.
`boolean removeAll (Collection<?> c)`	It removes all of this collection's elements that are also contained in the given collection.
`boolean retainAll (Collection<?> c)`	It keeps only the elements contained in the given collection.
`abstract int size()`	It gets the number of elements in this collection.
`Object[] toArray()`	It gets an array containing all the elements in this collection.
`<T> T[] toArray(T[ ] a)`	It gets an array that contains all the elements in this collection.
`String toString()`	It gets a string representation of this collection.

The *AbstractList* Class

The `AbstractList` class extends the `AbstractCollection` class. It is the foundation of other classes such as `ArrayList`, which supports dynamic arrays. Therefore, we'll refer `AbstractList` here for reference. The inheritance diagram for this class is as follows:

```
java.lang.Object
  |____java.util.AbstractCollection
        |____java.util.AbstractList
```

You'll find the field of the `AbstractList` class in Table 21.12, its constructor in Table 21.13, and its methods in Table 21.14:

Table 21.12: The field of the AbstractList class

Field	Does this
`protected int modCount`	It indicates the number of times this list has been modified.

Table 21.13: The constructor of the AbstractList class

Constructor	Does this
`protected AbstractList()`	It constructs an `AbstractList` object.

Table 21.14: Methods of the AbstractList class	
Method	**Does this**
`void add (int index, E element)`	It inserts the given element into this list.
`boolean add(E e)`	It adds the given element to the end of this list.
`boolean addAll (int index, Collection<? extends E> c)`	It inserts the elements in the given collection into this list.
`void clear()`	It removes all the elements from this list.
`boolean equals(Object o)`	It compares the given object with this list for equality.
`abstract Object get (int index)`	It gets the element at the given position in this list.
`int hashCode()`	It gets the hashcode value for this list.
`int indexOf(Object o)`	It gets the index of the first occurrence of the given element in this list, or -1 if this list does not contain the element.
`Iterator iterator()`	It gets an iterator over the elements in this list in proper sequence.
`int lastIndexOf(Object o)`	It gets the index of the last occurrence of the given element in this list, or -1 if this list does not contain the element.
`ListIterator listIterator()`	It gets a list iterator over the elements in this list (in proper sequence).
`ListIterator listIterator (int index)`	It gets a list iterator of the elements in this list (in proper sequence), starting at the given position in this list.
`E remove(int index)`	It removes the element at the given position in this list.
`protected void removeRange (int fromIndex, int toIndex)`	It removes from this list all the elements whose index is between `fromIndex` and `toIndex`.
`E set(int index, E element)`	It replaces the element at the given position in this list with the new element.
`List subList(int fromIndex, int toIndex)`	It gets a view of the portion of this list between the specified `fromIndex` (inclusive) and `toIndex` (exclusive).

The *AbstractSequentialList* Class

The `AbstractSequentialList` class is derived from the `AbstractList` class. It is the foundation of classes such as `LinkedList`. Let's look at the inheritance diagram for the `AbstractSequentialList` class:

```
java.lang.Object
  |____java.util.AbstractCollection
        |____java.util.AbstractList
              |____java.util.AbstractSequentialList
```

You'll find the constructor of the `AbstractSequentialList` class in Table 21.15 and its methods in Table 21.16:

Table 21.15: The constructor of the AbstractSequentialList class	
Constructor	**Does this**
`protected AbstractSequentialList()`	It constructs an `AbstractSequentialList` object.

Table 21.16: Methods of the AbstractSequentialList class	
Method	**Does this**
`void add (int index, E element)`	It inserts the given element at the given position in this list (optional operation).
`boolean addAll(int index, Collection<? extends E> c)`	It inserts all the elements in the given collection into this list at the specified position (optional operation).
`Object get(int index)`	It gets the element at the given position in this list.

Table 21.16: Methods of the AbstractSequentialList class	
Method	**Does this**
`Iterator<E> iterator()`	It gets an iterator over the elements in this list.
`abstract ListIterator<E> listIterator(int index)`	It gets a list iterator over the elements in this list.
`E remove(int index)`	It removes the element at the given position in this list (optional operation).
`E set (int index, E element)`	It replaces the element at the given position in this list with the specified element (optional operation).

The *ArrayList* Class

The Novice Programmer appears and says, "Good Lord! My code stores phone numbers in an array, but we are never sure whether the user will store 3 phone numbers there or 10,000, so I have to set up an array with enough space for 10,000 numbers." "Unless," you say, "you use an `ArrayList` object, which is a dynamic array that can grow at runtime." The NP says, "Wow!"

The `ArrayList` class is an array class that can grow or shrink at runtime. Note that arrays of this class must hold objects, not just simple data types. Here's the inheritance diagram for this class:

```
java.lang.Object
  |____java.util.AbstractCollection
        |____java.util.AbstractList
              |____java.util.ArrayList
```

You'll find the constructors of the `ArrayList` class in Table 21.17 and its methods in Table 21.18:

Table 21.17: Constructors of the ArrayList class	
Constructor	**Does this**
`ArrayList()`	It constructs an `ArrayList` object.
`ArrayList(Collection<? extends E> c)`	It constructs a list containing the elements of the given collection.
`ArrayList(int initialCapacity)`	It constructs an empty list with the given initial capacity.

Table 21.18: Methods of the ArrayList class	
Method	**Does this**
`void add(int index, E element)`	It inserts the given element at the given position in this list.
`boolean add(E e)`	It adds the given element to the end of this list.
`boolean addAll (Collection<? extends E> c)`	It adds all the elements in the given collection to the end of this list in the order that they are returned by the specified collection's Iterator.
`boolean addAll (int index, Collection<? extends E> c)`	It inserts all the elements in the given collection into this list, starting at the given position.
`void clear()`	It removes all the elements from this list.
`Object clone()`	It gets a copy of this `ArrayList` instance.
`boolean contains (Object elem)`	It returns True if this list contains the given element.
`void ensureCapacity (int minCapacity)`	It increases the capacity of this `ArrayList` instance.
`void forEach(Consumer<? super E> action)`	It performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
`Object get(int index)`	It gets the element at the given position in this list.
`int indexOf(Object elem)`	It returns the index of the first occurrence of the given element in this list, or -1 if this list does not contain the element.
`boolean isEmpty()`	It returns true if this list has no elements.

Table 21.18: Methods of the ArrayList class

Method	Does this
`int lastIndexOf (Object elem)`	It gets the index of the last occurrence of the given element in this list, or -1 if this list has no element.
`E remove(int index)`	It removes the element at the given position in this list.
`boolean remove(Object elem)`	It removes the first occurrence of the specified element from this list, if it is present.
`protected void removeRange (int fromIndex, int toIndex)`	It removes from this list all the elements whose index is between `fromIndex` and `toIndex`.
`E set(int index, E element)`	It replaces the element at the given position in this list with the given element.
`int size()`	It gets the number of elements in this list.
`void sort(Comparator<? super E> c)`	It sorts this list according to the order induced by the specified Comparator.
`Spliterator<E> spliterator()`	It creates a late-binding and fail-fast Spliterator over the elements in this list.
`List<E> subList(int fromIndex, int toIndex)`	It returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
`Object[] toArray()`	It gets an array containing all the elements in this list in proper sequence(from first to last element).
`<T> T[] toArray(T[] a)`	It gets an array containing all the elements in this list in proper sequence(from first to last element); the runtime type of the returned array is that of the specified array.
`void trimToSize()`	It trims the capacity to be the list's current size.

You can add elements to an `ArrayList` object with the `add()` method, get an element at a certain index with the `get()` method, and remove elements with the `remove()` method.

Let's consider an example in which we add elements to an array at runtime. Remember we are using two forms of the `add()` method: general form and the form that allows to add an element at a specific location(index). We are also using the `remove()` method to remove one element. The `Iterator` interface is defined in the `java.util` package. This `Iterator` interface has three methods—`hasNext()`, `next()`, and `remove()`. These methods are all you need to access each element of a collection, as you see here:

```
import java.util.*;
class Arraylist
{
  public static void main(String args[])
  {
        ArrayList<String> arraylist = new ArrayList<String>();
        arraylist.add("Item 0");
        arraylist.add("Item 2");
        arraylist.add("Item 3");
        arraylist.add("Item 4");
        arraylist.add("Item 5");
        arraylist.add("Item 6");
        arraylist.add(1, "Item 1");

        System.out.println("\nUsing the add method");
        System.out.println(arraylist);

        arraylist.remove("Item 5");

        System.out.println(arraylist);
        System.out.println("\nUsing the Iterator interface");
        String s;
        Iterator e = arraylist.iterator();
        while (e.hasNext())
        {
```

```
                    s = (String)e.next();
                    System.out.println(s);
                }
            }
        }
```

Here's the output of this example:

```
C:\>java Arraylist
Using the add method
[Item 0, Item 1, Item 2, Item 3, Item 4, Item 5, Item 6]
[Item 0, Item 1, Item 2, Item 3, Item 4, Item 6]

Using the Iterator interface
Item 0
Item 1
Item 2
Item 3
Item 4
Item 6
```

NOTE

Generics is on the leading edge of Object-Oriented Programming. You can get by without knowing anything about generics, but it is better to just have an idea about them so as to learn how ArrayList and LinkedList classes use the new generics feature.

The *ArrayDeque* Class

The `ArrayDeque` class was added in Java SE 6. It extends `AbstractCollection` and implements the `Deque` interface. It does not add methods on its own. The `ArrayDeque` class forms a dynamic array which has no limitations on capacity. The `Deque` interface supports implementation that limits capacity, but does not require such restrictions.

The `ArrayDeque` class is a generic class and its inheritance diagram is as follows:

```
java.lang.Object
  |____java.util.AbstractCollection
        |____java.util.ArrayDeque
```

You'll find the constructors of the `ArrayDeque` class in Table 21.19 and its methods in Table 21.20:

Table 21.19: Constructors of the ArrayDeque class

Constructor	Does this
`ArrayDeque()`	It constructs an `ArrayDeque` object.
`ArrayDeque(Collection<? extends E> c)`	It constructs a deque containing the elements of the given collection.
`ArrayDeque(int numElements)`	It constructs an empty deque with the given initial capacity.

Table 21.20: Methods of the ArrayDeque class

Method	Does this
`boolean add(E e)`	It adds the given element to the end of this deque.
`void addFirst(E e)`	It adds the given element to the front of this deque.
`void addLast (E e)`	It inserts the given element at the end of this deque.
`void clear()`	It removes all the elements from this deque.
`ArrayDeque<E> clone()`	It gets a copy of this `ArrayDeque` instance.
`boolean contains (Object elem)`	It returns True if this deque contains the given element.
`Iterator<E> descendingIterator()`	It returns an iterator over the elements in this deque in reverse sequential order.
`E element()`	It retrieves, but does not remove, the head of the queue represented by this deque.
`E getFirst()`	It retrieves, but does not remove, the first elements of this deque.

Table 21.20: Methods of the ArrayDeque class

Method	Does this
`E getLast()`	It retrieves, but does not remove, the last elements of this deque.
`boolean isEmpty()`	It returns true if this deque has no elements.
`Iterator<E> iterator()`	It returns an iterator over the elements in this deque.
`boolean offer(E e)`	It inserts the specified element at the end of this deque.
`boolean offerFirst(E e)`	It inserts the specified element at the front of this deque.
`boolean offerLast(E e)`	It inserts the specified element at the end of this deque.
`E peek()`	It retrieves, but does not remove, the head of the queue represented by this deque, or return null if this deque is empty.
`E peekFirst()`	It retrieves, but does not remove, the first element of this deque, or return null if this deque is empty.
`E peekLast()`	It retrieves, but does not remove, the last element of this deque, or return null if this deque is empty.
`E poll()`	It retrieves and removes, the head of the queue represented by this deque, or return null if this deque is empty.
`E pollFirst()`	It retrieves and removes the first element of this deque or return null if this deque is empty.
`E pollLast()`	It retrieves and removes the last element of this deque or return null if this deque is empty.
`E pop()`	It pops an element from the stack represented by this deque.
`void push(E e)`	It pushes an element from the stack represented by this deque.
`E remove()`	It retrieves and removes the head of the queue represented by this deque.
`E removeFirst()`	It retrieves and removes the first element of this deque.
`E removeLast()`	It retrieves and removes the last element of this deque.
`boolean removeFirstOccurrence (Object o)`	It removes the last occurrence of the specified element in this deque.
`int size()`	It returns the number of elements in this deque.
Spliterator<E> `spliterator()`	It creates a late-binding and fail-fast Spliterator over the elements in this deque.
`Object toArray()`	It returns an array containing all of the elements in this deque in proper sequence(from first to last element).
`<T> T[] toArray(T[] a)`	It returns an array containing all of the elements in this deque in proper sequence(from first to last element); the runtime type of the returned array is that of the specified array.

To build a linked list, you can use the `add()`, `addFirst()`, and `addLast()` methods; to get an element at a certain index, you can use the `get()`, `getFirst()`, and `getLast()` methods; to set an element's value, you can use the `set` method; and to remove an element, you can use the `remove()`, `removeFirst()`, and `removeLast()` methods.

Here's an example which demonstrates `ArrayDeque` by using it to create a stack:

```
import java.util.*;
class ArrayDequeDemo { public static void main(String args[])
{
        ArrayDeque<String> arrdeque= new ArrayDeque<String>();
        arrdeque.push("Item 0"); arrdeque.push("Item 2");
        arrdeque.push("Item 3");
        arrdeque.push("Item 4");
        arrdeque.push("Item 5");
        arrdeque.push("Item 6");
```

```
            System.out.println("Popping the stack: ");
            while (arrdeque.peek() != null)
            {
                    System.out.println(arrdeque.pop() + " ");
            }
            System.out.println();
      }
    }
```

Here's the output of this example:

```
C:\>java ArrayDequeDemo
Popping the stack:
Item 6
Item 5
Item 4
Item 3
Item 2
Item 0
```

The *LinkedList* Class

The Programming Correctness expert arrives and says, "Well, Java has many features of C++, but what about support for programming constructs such as linked lists?" "No problem," you say.

The `LinkedList` class supports linked lists in which a record of items is maintained in these lists. With the use of such a list, you can move one element to the next using an iterator, which you'll learn in this chapter. The inheritance diagram for the `LinkedList` class is as follows:

```
java.lang.Object
  |____java.util.AbstractCollection
        |____java.util.AbstractList
              |____java.util.AbstractSequentialList
                    |____java.util.LinkedList
```

You'll find the constructors of the `LinkedList` class in Table 21.21 and its methods in Table 21.22:

Table 21.21: Constructors of the LinkedList class

Constructor	Does this
`LinkedList()`	It constructs a `LinkedList` object.
`LinkedList(Collection<? extends E> c)`	It constructs a list containing the elements of the given collection.

Table 21.22: Methods of the LinkedList class

Method	Does this
`void add(int index, E element)`	It inserts the given element at the given position.
`boolean add(E e)`	It adds the given element to the end of this list.
`boolean addAll (Collection<? extends E> c)`	It adds all the elements in the given collection to the end of this list.
`boolean addAll(int index, Collection<? extends E> c)`	It inserts all the elements in the given collection into this list.
`void addFirst(E e)`	It inserts the specified element at the beginning of this list.
`void addLast(E e)`	It adds the specified element to the end of this list.
`void clear()`	It removes all the elements from this list.
`Object clone()`	It gets a copy of this `LinkedList` object.
`boolean contains(Object o)`	It returns True if this list contains the given element.
`Iterator<E> descendingIterator()`	It gets an iterator over the elements in this deque in reverse sequential order.
`E element()`	It gets, but does not remove, the head (first element) of this list.
`E get(int index)`	It gets the element at the given position in this list.

Table 21.22: Methods of the LinkedList class

Method	Does this
`E getFirst()`	It gets the first element in this list.
`E getLast()`	It gets the last element in this list.
`int indexOf(Object o)`	It gets the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`int lastIndexOf(Object o)`	It gets the index of the last occurrence of the specified element in this list or -1, if this list does not contain the element.
`ListIterator<E> listIterator(int index)`	It gets a list iterator of the elements in this list, starting at the given position in the list.
`boolean offer(E e)`	It appends the specified elements as the tail (last element) of this list.
`boolean offerFirst (E e)`	It inserts the specified element at the front of this list.
`boolean offerLast(E e)`	It inserts the specified element at the end of this list.
`E peak()`	It gets the element, but does not remove, the head (first element) of this list.
`E peakFirst()`	It gets the element, but does not remove, the first element of this list, or returns null if this list is empty.
`E peakLast()`	It gets the element, but does not remove, the last element of this list, or returns null if this list is empty.
`E poll()`	It gets and removes the head (first element) of this list.
`E pollFirst()`	It gets and removes the first element of this list, or returns null if this list is empty.
`E pollLast()`	It gets and remove the last element of this list or returns null if this list is empty.
`E pop()`	It pops an element from the stack represented by this list.
`void push(E e)`	It pushes an element onto the stack represented by this list.
`E remove(int index)`	It removes the element at the given position in this list.
`boolean remove(Object o)`	It removes the first occurrence of the given element in this list.
`E removeFirst()`	It removes and returns the first element from this list.
`boolean removeFirstOccurrence (Object e)`	It removes the first occurrence of the specified element in this list.
`E removeLast()`	It removes and returns the last element from this list.
`boolean removeLastOccurrence(Object e)`	It removes the last occurrence of the specified element in this list.
`E set(int index, E element)`	It replaces the element at the given position in this list with the given element.
`int size()`	It gets the number of elements in this list.
Spliterator<E> HYPERLINK "http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html" \l "spliterator--" spliterator`()`	It creates a late-binding and fail-fast Spliterator over the elements in this list.
`Object[] toArray()`	It gets an array containing all the elements in this list in the correct order.
`<T> T[] toArray (T[] a)`	It gets an array containing all the elements in this list in the correct order.

To build a linked list, you can use the `add()`, `addFirst()`, and `addLast()` methods; to get an element at a certain index, you can use the `get()`, `getFirst()`, and `getLast()` methods; to set an element's value, you can use the `set()` method; and to remove an element, you can use the `remove()`, `removeFirst()`, and `removeLast()` methods.

Here's an example: We are simply building a linked list, adding some elements, and then removing some elements:

```
import java.util.*;

class Linkedlist
{
  public static void main(String args[])
  {
        LinkedList<String> linkedlist1 = new LinkedList<String>();
        linkedlist1.add("Item 2");
        linkedlist1.add("Item 3");
        linkedlist1.add("Item 4");
        linkedlist1.add("Item 5");
        linkedlist1.addFirst("Item 0");
        linkedlist1.addLast("Item 6");
        linkedlist1.add(1, "Item 1");
        System.out.println(linkedlist1);
        linkedlist1.remove("Item 6");
        System.out.println(linkedlist1);
        linkedlist1.removeLast();

        System.out.println(linkedlist1);
        System.out.println("\nUpdating linked list items");

        linkedlist1.set(0, "Red");
        linkedlist1.set(1, "Blue");
        linkedlist1.set(2, "Green");
        linkedlist1.set(3, "Yellow");
        linkedlist1.set(4, "Purple");
        System.out.println(linkedlist1);
  }
}
```

Here's the output of this example:

```
[Item 0, Item 1, Item 2, Item 3, Item 4, Item 5, Item 6]
[Item 0, Item 1, Item 2, Item 3, Item 4, Item 5]
[Item 0, Item 1, Item 2, Item 3, Item 4]

Updating linked list items
[Red, Blue, Green, Yellow, Purple]
```

NOTE

Generics is on the leading edge of Object-Oriented Programming. You can move further without knowing much about generics, but it is preferable to have an idea about them so as to get utter knowledge on how the ArrayList and LinkedList classes use the new generic features.

The *Generic* Class

In case of the `ArrayList` and `LinkedList` classes, the compiler does not allow us to add the wrong type of data because it took the benefit of a new feature of Java 1.5 called Generics.

This feature will be used in the `LinkedList` class to implement a particular type of collection stacks. In this case, the items are always added to the beginning of the list and also retrieved from the beginning of the list.

Let's begin by designing a `GenStack` class that uses a `LinkedList` to execute a stack:

```
import java.util.*;

public class GenStack<E>
{
  private LinkedList<E> list = new LinkedList<E>();
  public void push(E item) { list.addFirst(item); }
  public E pop() { return list.poll(); }
  public E peek() { return list.peek(); }
  public boolean hasItems() { return !list.isEmpty(); }
  public int size() { return list.size(); }
}
```

The formal type parameter <E> specifies the type of elements that are stored in the list. Compile the code now. Now we'll make an example that works with the GenStack class:

```
import java.util.*;

public class StackTest
{
  public static void main(String[] args)
  {
        GenStack<String> gs = new GenStack<String>();
        System.out.println("Pushing Tomato, Potato , Onion and Capsicum
        into the stack...");
        gs.push("Tomato"); gs.push("Potato "); gs.push("Onion ");
        gs.push("Capsicum ");   System.out.println("...Done...\n");
        System.out.println("The stack contains" + gs.size() + " items in the
        stack and the top item is " + gs.peek() + ".\n");
        System.out.println("When the " + gs.size() + " items in the stack
        are Popped, they are displayed as: ");
        while (gs.hasItems()) { System.out.println(gs.pop()); }
        System.out.println("\nNow there are " + gs.size() + " item(s) in
        the stack and the top item is " + gs.peek() + ".\n"); }
}
```

When the items are popped off the stack, they come out in a reverse manner—that's a normal behavior of stack based on LIFO (Last-in First-out) principle. The result of the example is as follows:

```
Pushing Tomato, Potato, Onion and Capsicum into the stack...
...Done...

The stack contains  4 items in the stack and the top item is Capsicum.

When the 4 items in the stack are Popped, they are displayed as:
Capsicum
Onion
Potato
Tomato

Now there are 0 item(s) in the stack and the top item is null.
```

The *HashSet* Class

A set is defined as a group of unique elements, and the HashSet class supports sets that use a hash internally for storage. Because this class uses a hash internally, methods such as contains(), remove(), add(), and size() always take the same amount of time to execute. The inheritance diagram for this class is as follows:

```
java.lang.Object
  |____java.util.AbstractCollection
        |____java.util.AbstractSet
              |____java.util.HashSet
```

You'll find the constructors of the HashSet class in Table 21.23 and its methods in Table 21.24:

Table 21.23: Constructors of the HashSet class

Constructor	Does this
HashSet()	It constructs a new, empty set.
HashSet(Collection<? extends E> c)	It constructs a new set containing the elements in the given collection.
HashSet(int initialCapacity)	It constructs a new, empty set with the given initial capacity and default load factor (0.75).
HashSet (int initialCapacity, float loadFactor)	It constructs a new, empty set with the given initial capacity and the given load factor.

Table 21.24: Methods of the HashSet class

Method	Does this
boolean add(E e)	It adds the given element to this set if it's not already present.

Table 21.24: Methods of the HashSet class	
Method	**Does this**
`void clear()`	It removes all the elements from this set.
`Object clone()`	It gets a copy of this `HashSet` instance.
`boolean contains(Object o)`	It returns True if this set contains the given element.
`boolean isEmpty()`	It returns True if this set contains no elements.
`Iterator<E> iterator()`	It gets an iterator over the elements in this set.
`boolean remove(Object o)`	It removes the given element from this set if it's present.
`int size()`	It gets the number of elements in this set.
Spliterator<E> `spliterator()`	It creates a late-binding and fail-fast Spliterator over the elements in this set.

Here's an example in which we add elements to a set and then print out that set:

```
import java.util.*;

class Hashset
{
  public static void main(String args[])
  {
        HashSet<String> hashset1 = new HashSet<String>();
        hashset1.add("Item 0");
        hashset1.add("Item 1");
        hashset1.add("Item 2");
        hashset1.add("Item 3");
        hashset1.add("Item 4");
        hashset1.add("Item 5");
        hashset1.add("Item 6");
        System.out.println(hashset1);
  }
}
```

One thing to note about hashes is that you can't guarantee the order in which elements are stored internally. In fact, when you print out this set, you can see that all elements are stored in exactly reverse order:

```
C:\>java Hashset
[Item 0, Item 4, Item 3, Item 2, Item 1, Item 6, Item 5]
```

The *TreeSet* Class

The `TreeSet` class uses a tree construct for internal storage to implement a set, which means that access times are fast. The elements of the `TreeSet` class are stored in a sorted order and its inheritance diagram is as follows:

```
java.lang.Object
  |____java.util.AbstractCollection
        |____java.util.AbstractSet
              |____java.util.TreeSet
```

You'll find the constructors for the `TreeSet` class in Table 21.25 and its methods in Table 21.26:

Table 21.25: Constructors of the TreeSet class	
Constructor	**Does this**
`TreeSet()`	It constructs a new, empty tree set, sorting according to the natural ordering of its elements.
`TreeSet(Collection<? extends E> c)`	It constructs a new tree set containing the elements in the given collection, sorting according to the natural ordering of its elements.
`TreeSet(Comparator<? super E> comparator)`	It constructs a new, empty tree set, sorted according to the given comparator.
`TreeSet(SortedSet<E> s)`	It constructs a new tree set containing the same elements and using the same ordering as the given sorted set.

Table 21.26: Methods of the TreeSet class	
Method	**Does this**
`boolean add(E e)`	It adds the given element to this set.
`boolean addAll (Collection<? extends E> c)`	It adds all the elements in the given collection.
`E ceiling(E e)`	It returns the least element in this set greater than or equal to the given element, or null if there is no such element.
`void clear()`	It removes all the elements from this set.
`Object clone()`	It gets a shallow copy of this `TreeSet` instance.
`Comparator <? super E> comparator()`	It gets the comparator used to order the element in this set or null if this set uses the natural ordering.
`boolean contains(Object o)`	It returns True if this set contains the given element.
`Iterator<E> descendingIterator()`	It returns an iterator over the elements in this set in descending order.
`NavigableSet<E> descendingSet()`	It returns a reverse order view of the elements contained in this set.
`E first()`	It gets the first (lowest) element currently in this set.
`E floor(E e)`	It gets the greatest element in this set less than or equal to the given element, or null if there is no such element.
`SortedSet<E> headset(E e)`	Equivalent to `navigableHeadSet` (Object e) `SortedSet` interface.
`NavigableSet<E> headSet(E toElement, boolean inclusive)`	It returns a view of the portion of this set whose elements are less than (or equal to, if inclusive is true) toElement.
`E higher(E e)`	It returns the least element in this set, which is greater than the given element, or null if there is no such element.
`boolean isEmpty()`	It returns True if this set contains no elements.
`Iterator<E> iterator()`	It gets an iterator over the elements in this set in ascending order.
`E last()`	It gets the last (highest) element currently in this set.
`E lower(E e)`	It returns the greatest element in this set strictly less than the given element, or null if there is no such element.
`NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)`	It returns a view of the portion of this set whose elements range from fromElement to toElement.
`SortedSet<E> subSet(E fromElement, E toElement)`	It returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
`SortedSet<E> tailSet(E fromElement)`	It returns a view of the portion of this set whose elements are greater than or equal to fromElement.
NavigableSet<E> `tailSet(E fromElement, boolean inclusive)`	It returns a view of the portion of this set whose elements are greater than (or equal to, if inclusive is true) fromElement.
`E pollFirst()`	It gets and removes the first element, or returns null if this set is empty.
`E pollLast()`	It gets and removes the last element, or returns null if this set is empty.
`boolean remove(E e)`	It removes the given element from this set if it's present.
`int size()`	It gets the number of elements in this set.

The `TreeSet` class just implements a set using a tree for internal storage. A difference from the `HashSet` class is that elements in a `TreeSet` object are stored in sorted order. Here's the same example as the previous one, where we stored elements in a `HashSet` object; however, this time we are using a `TreeSet` object:

```
import java.util.*;

class Treeset
{
  public static void main(String args[])
```

```
    {
          TreeSet<String> treeset1 = new TreeSet<String>();
          treeset1.add("Item 0");
          treeset1.add("Item 1");
          treeset1.add("Item 2");
          treeset1.add("Item 3");
          treeset1.add("Item 4");
          treeset1.add("Item 6");
          treeset1.add("Item 5");
          System.out.println(treeset1);
    }
}
```

Here's the output of this example (note that, unlike the `HashSet` example in the previous solution, the `TreeSet` elements are sorted in ascending order):

```
C:\>java Treeset
[Item 0, Item 1, Item 2, Item 3, Item 4, Item 5, Item 6]
```

In fact, you can specify the sorting order for the `TreeSet` objects—see the next solution for the details.

Using the **Comparator** Interface

The Novice Programmer appears and says, "I like the speed of `TreeSet` objects; on the other hand, sometimes I want to find out what's in those objects and sort the elements in a particular order." "No problem," you say, "just build up a `Comparator` class." "How's that?" the NP asks.

You can use a comparator to find the sorting order used in the `TreeSet` objects; for example, with the implementation of the `Comparator` interface. The methods of this interface are provided in Table 21.27:

Table 21.27: Methods of the Comparator interface

Method	Does this
`int compare(Object obj1, Object obj2)`	It compares its two arguments.
`boolean equals(Object obj)`	It specifies whether another object is equal to this comparator.

The `compare()` method compares two objects, `obj1` and `obj2`, and returns -1 if `obj1` is less than `obj2`; 0 if they are equal; and 1 if `obj1` is greater than `obj2`. By overriding this method, you can support a custom sorting order.

Here's an example in which we sort a `TreeSet` object in ascending order—except for one element, Item 3, which we always want to appear at the beginning of the set. Here's what this looks like in code:

```
import java.util.*;

class ComparatorDemo
{
  @SuppressWarnings("unchecked")
  public static void main(String args[])
  {
        TreeSet<String> treeset = new TreeSet<String>(new NewComparator());
        treeset.add("Item 0");
        treeset.add("Item 1");
        treeset.add("Item 2");
        treeset.add("Item 3");
        treeset.add("Item 4");
        treeset.add("Item 5");
        treeset.add("Item 6");
        Iterator iterator = treeset.iterator();
        while(iterator.hasNext()) { System.out.println(iterator.next()); }
  }
}
class NewComparator implements Comparator {
  public int compare(Object obj1, Object obj2) {
        if (((String) obj1).equals("Item 3")) return -1;
```

```
      return ((String) obj1).compareTo((String) obj2); }
   }
```

Here's the output of this example. Note that Item 3 comes first:

```
C:\>java ComparatorDemo
Item 3
Item 0
Item 1
Item 2
Item 4
Item 5
Item 6
```

Using the **Iterator** Interface

"Hmm," says the Novice Programmer, "I want to loop over all the elements in a list. How do I do that?" "With an iterator," you say. "OK," says the NP, "but what's an iterator?"

The methods of the `Iterator` interface can be used to move through an accumulation using the `next()` method, allowing you to cycle through the elements of the collection. The methods of the `Iterator` interface are shown in Table 21.28:

Table 21.28: Methods of the Iterator interface

Method	Does this
`boolean hasNext()`	It returns True if the iteration has more elements.
`Object next()`	It gets the next element in the iteration.
`void remove()`	It removes the last element returned.

Here's an example in which we print out each element in a linked list from first to last using an iterator and the methods `next()` and `hasNext()` (which returns True if the collection has a next element):

```
import java.util.*;
class IteratorDemo {
public static void main(String args[]) {
LinkedList<String> linkedlist = new LinkedList<String>();
linkedlist.add("Item 0");
linkedlist.add("Item 1");
linkedlist.add("Item 2");
linkedlist.add("Item 3");
linkedlist.add("Item 4");
linkedlist.add("Item 5");
linkedlist.add("Item 6");
Iterator<String> iterator = linkedlist.iterator();
while(iterator.hasNext()) { System.out.println(iterator.next()); }
}
}
```

Here's the output of this example:

```
C:\>java IteratorDemo
Item 0
Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
```

Besides iterators, you can also use list iterators. See the next solution.

Using the **ListIterator** Interface

"Well," says the Novice Programmer, "iterators are fine, but I want to work through a collection backwards. No way, huh?" "No problem," you say, "just use a list iterator." The NP says, "Tell me more!"

Whereas iterators can only work forward through a list using the `next()` method, list iterators also support the `previous()` method, so they can work backwards through lists. This is particularly useful for linked lists, where each element often must know about the next and previous elements (called a *bidirectional linked list*). This is handy, for example, when you are implementing a buffer that can grow or shrink. You'll find the methods of the `ListIterator` interface in Table 21.29:

Table 21.29: Methods of the ListIterator interface

Method	Does this
`void add(E e)`	It inserts the given element into the list.
`boolean hasNext()`	It returns True if this list iterator has more elements in the forward direction.
`boolean hasPrevious()`	It returns True if this list iterator has more elements in the reverse direction.
`E next()`	It gets the next element in the list.
`int nextIndex()`	It gets the index of the element that would be returned by a subsequent call to `next()`.
`E previous()`	It gets the previous element in the list.
`int previousIndex()`	It gets the index of the element that would be returned by a subsequent call to `previous`.
`void remove()`	It removes from the list the last element that was returned by `next` or `previous`.
`void set(E e)`	It replaces the last element returned by `next` or `previous` with the given element.

Here's an example in which we print a linked list backward:

```
import java.util.*;

class Listiterator
{
  public static void main(String args[])
  {
        LinkedList<String> linkedlist = new LinkedList<String>();
        linkedlist.add("Item 0");
        linkedlist.add("Item 1");
        linkedlist.add("Item 2");
        linkedlist.add("Item 3");
        linkedlist.add("Item 4");
        linkedlist.add("Item 5");
        linkedlist.add("Item 6");
        ListIterator<String> listiterator = linkedlist.listIterator();
        while(listiterator.hasNext())
        {
                listiterator.set("This is " + listiterator.next()); }
                while(listiterator.hasPrevious()) {
                        System.out.println(listiterator.previous()); }
  }
}
```

Here's the output of this example:

```
C:\>java Listiterator
This is Item 6
This is Item 5
This is Item 4
This is Item 3
This is Item 2
This is Item 1
This is Item 0
```

Here's another example in which we modify and print an arraylist backwards:

```
import java.util.*;
class ListArray {
```

```
        public static void main(String args[]) {
                ArrayList<String> arr = new ArrayList<String>();
                arr.add("Item 1");
                arr.add("Item 2");
                arr.add("Item 3");
                arr.add("Item 4");
                arr.add("Item 5");
                arr.add("Item 6");
                System.out.print("The contents of arr:\n");
                Iterator<String> iter = arr.iterator();
                while(iter.hasNext()) {
                        String itrTxt = iter.next();
                        System.out.print(itrTxt + " "); }
                System.out.println();
                ListIterator<String> litarr = arr.listIterator();
                while(litarr.hasNext()) {
                        String itrTxt = litarr.next();
                        litarr.set(itrTxt + " |_|"); }
                System.out.print("\nContents of arr modified:\n");
                iter = arr.iterator();
                while(iter.hasNext()) {
                        String itrTxt = iter.next();
                        System.out.print(itrTxt + " "); }
                System.out.println();
                System.out.print("\nContents listed backwards:\n");
                while(litarr.hasPrevious()) {
                        String itrTxt = litarr.previous();
                        System.out.print(itrTxt + " "); }
                System.out.println(); }
    }
```

Here's the result of this example:

```
The contents of arr:
Item 1 Item 2 Item 3 Item 4 Item 5 Item 6

Contents of arr modified:
Item 1 |_| Item 2 |_| Item 3 |_| Item 4 |_| Item 5 |_| Item 6 |_|

Contents listed backwards:
Item 6 |_| Item 5 |_| Item 4 |_| Item 3 |_| Item 2 |_| Item 1 |_|
```

Using the **AbstractMap** Class

The `AbstractMap` class is the base of the map classes in Java, so we'll give a quick glance to refer it. The inheritance diagram for the `AbstractMap` class is as follows:

```
java.lang.Object
  |____java.util.AbstractMap
```

You'll find the constructor of the `AbstractMap` class in Table 21.30 and its methods in Table 21.31:

Table 21.30: The constructor of the AbstractMap class

Constructor	Does this
`protected AbstractMap()`	It constructs an `AbstractMap` object.

Table 21.31: Methods of the AbstractMap class

Method	Does this
`void clear()`	It removes all mappings.
`boolean containsKey (Object key)`	It returns True if this map contains a mapping for the given key.
`boolean containsValue (Object value)`	It returns True if this map maps one or more keys to the specified.

Table 21.31: Methods of the AbstractMap class	
Method	**Does this**
`abstract Set<Map.Entry<K,V>> entrySet()`	It returns a Set.
`boolean equals(Object o)`	It compares the given object with this map for equality.
`V get(Object key)`	It gets the value to which the specified key is mapped or null if this map contains no mapping for the given key.
`int hashCode()`	It gets the hashcode value for this map.
`boolean isEmpty()`	It returns True if this map contains no key/value mappings.
`Set<K> keySet()`	It returns a Set.
`V put(K key, V value)`	It associates the given value with the given key in this map.
`void putAll(Map<? extends K,? extends V> m)`	It copies all the mappings from the given map to this map.
`Object remove(Object key)`	It removes the mapping for a key from this map, if it is present.
`int size()`	It gets the number of key/value mappings in this map.
`String toString()`	It gets a string representation of this map.
`Collection<V> values()`	It returns a Collection.

Using the **HashMap** Class

"I'm writing a phonebook application," the Novice Programmer says, "and it's hard to keep track of the people in it by converting them into indexes in an array. Wouldn't it be great if I could access an array using strings such as people's names instead of numeric indexes?" "You can." you say, "Just use a map."

Maps make it possible to store data as key/value pairs, where both the key and the values are the objects. For example, creation of a map indexed with strings and not numbers. The inheritance diagram for the `HashMap` class is as follows:

```
java.lang.Object
  |____java.util.AbstractMap
        |____java.util.HashMap
```

You'll find the constructors for the `HashMap` class in Table 21.32 and its methods in Table 21.33:

Table 21.32: Constructors of the HashMap class	
Constructor	**Does this**
`HashMap()`	It constructs a new, empty map.
`HashMap(int initialCapacity)`	It constructs a new, empty map with the given initial capacity.
`HashMap(int initialCapacity, float loadFactor)`	It constructs a new, empty map with the given initial capacity and the given load factor.
`HashMap (Map<? extends K,? extends V> m)`	It constructs a new map with the same mappings as the given map.

Table 21.33: Methods of the HashMap class	
Method	**Does this**
`void clear()`	It removes all of the mappings from this map.
`Object clone()`	It gets a copy of this `HashMap` instance.
`boolean containsKey (Object key)`	It returns True if this map contains a mapping for the given key.
`boolean containsValue (Object value)`	It returns True if this map maps one or more keys to the given value.
`Set<Map.Entry<K,V>> entrySet()`	It returns a Set.

Table 21.33: Methods of the HashMap class	
Method	**Does this**
`void forEach(BiConsumer<? super K,? super V> action)`	It performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
`V get(Object key)`	It gets the value to which the given key is mapped, or null if this map contains no mapping for the key.
`V getOrDefault(Object key, V defaultValue)`	It returns the value to which the specified key is mapped or defaultValue if this map contains no mapping for the key.
`boolean isEmpty()`	It returns True if this map contains no key/value mappings.
`Set<K> keySet()`	It returns a `Set`.
`V put(K key, V value)`	It associates the given value with the given key in this map.
`void putAll (Map<? extends K,? extends V> m)`	It copies all the mappings from the given map to this map.
`V remove(Object key)`	It removes the mapping for the specified key from this map if present.
`boolean remove(Object key, Object value)`	It removes the entry for the specified key only if it is currently mapped to the specified value.
`boolean replace(K key, V oldValue, V newValue)`	It replaces the entry for the specified key only if currently mapped to the specified value.
`void replaceAll(BiFunction<? super K,? super V,? extends V> function)`	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
`int size()`	It gets the number of key/value mappings in this map.
`Collection<V> values()`	It gets a Collection.

The `put()` method is used to add a key/value pair to a map, and the `get()` method is used to retrieve a value, given a key. For example, the string `drink` will be left holding the text "root beer" after this example executes:

```
hashmap.put("drink", "root beer");
String drink = hashmap.get("drink");
```

You can also get a set corresponding to the map with the `entrySet()` method, and you can iterate over that set using the `Map.Entry` interface. Here's an example in which we create a map and then print out all the key/value pairs in the map:

```
import java.util.*;

class Hashmap
{
  public static void main(String args[])
  {
        HashMap<String, String> hashmap1 = new HashMap<String, String>();
        hashmap1.put("Item 0", "Value 0");
        hashmap1.put("Item 1", "Value 1");
        hashmap1.put("Item 2", "Value 2");
        hashmap1.put("Item 3", "Value 3");
        hashmap1.put("Item 4", "Value 4");
        hashmap1.put("Item 5", "Value 5");
        hashmap1.put("Item 6", "Value 6");
        Set set = hashmap1.entrySet();
        Iterator iterator = set.iterator();
        while(iterator.hasNext())
        {
                Map.Entry mapentry = (Map.Entry) iterator.next();
                System.out.println(mapentry.getKey() + "/" +
                mapentry.getValue());
        }
  }
}
```

Here's the output of this example:

```
C:\>java Hashmap
Item 0/Value 0
Item 4/Value 4
Item 3/Value 3
Item 2/Value 2
Item 1/Value 1
Item 6/Value 6
Item 5/Value 5
```

The following code iterates through the `System` property map [available through `System.getProperties()`], and displays each key-value pair:

```
import java.util.Map;
import java.util.Properties;
import java.util.Set;

public class SetEntry
{
  public static void main(String args[])
  {
        Properties props = System.getProperties();
        Set<Map.Entry<Object,Object>> entrySet = props.entrySet();
        for (Map.Entry entry: entrySet)
        {
                System.out.println(entry.getKey() + ": " +
                entry.getValue());
        }
  }
}
```

Notice the following line:

```
Set<Map.Entry<Object,Object>> entrySet = props.entrySet();
```

A set of `Map.Entry` objects is returned by the `entrySet()` method of the `Properties` object that doesn't require any casting operations for the `for-each` construct.

```
for (Map.Entry entry: entrySet) { . . . }
```

The result is as shown here:

```
C:\> java SetEntry
java.runtime.name: Java(TM) SE Runtime Environment
sun.boot.library.path: C:\Program Files\Java\jre1.8.0_31\bin
java.vm.version: 25.31-b07
java.vm.vendor: Oracle Corporation
java.vendor.url: http://java.oracle.com/
path.separator: ;
java.vm.name: Java HotSpot(TM) 64-Bit Server VM
file.encoding.pkg: sun.io
user.country: IN
user.script:
sun.java.launcher: SUN_STANDARD
sun.os.patch.level:
java.vm.specification.name: Java Virtual Machine Specification
user.dir: E:\Java8\chapter21
java.runtime.version: 1.8.0_31-b13
java.awt.graphicsenv: sun.awt.Win32GraphicsEnvironment
java.endorsed.dirs: C:\Program Files\Java\jre1.8.0_31\lib\endorsed
os.arch: amd64
java.io.tmpdir: C:\Users\DEEPAK~1\AppData\Local\Temp\
line.separator:
java.vm.specification.vendor: Oracle Corporation
user.variant:
os.name: Windows 8.1
sun.jnu.encoding: Cp1252
java.library.path: C:\ProgramData\Oracle\Java\javapath;C:\Windows\Sun\Java\bin;C
:\Windows\system32;C:\Windows;C:\ProgramData\Oracle\Java\javapath;C:\Program Fil
```

```
es\Java\jdk1.8.0_25\bin;;.
java.specification.name: Java Platform API Specification
java.class.version: 52.0
sun.management.compiler: HotSpot 64-Bit Tiered Compilers
os.version: 6.3
user.home: C:\Users\Deepak Sharma
user.timezone:
java.awt.printerjob: sun.awt.windows.WPrinterJob
file.encoding: Cp1252
java.specification.version: 1.8
java.class.path: .
user.name: Deepak Sharma
java.vm.specification.version: 1.8
sun.java.command: SetEntry
java.home: C:\Program Files\Java\jre1.8.0_31
sun.arch.data.model: 64
user.language: en
java.specification.vendor: Oracle Corporation
awt.toolkit: sun.awt.windows.WToolkit
java.vm.info: mixed mode
java.version: 1.8.0_31
java.ext.dirs: C:\Program Files\Java\jre1.8.0_31\lib\ext;C:\Windows\Sun\Java\lib
\ext
sun.boot.class.path: C:\Program Files\Java\jre1.8.0_31\lib\resources.jar;C:\Prog
ram Files\Java\jre1.8.0_31\lib\rt.jar;C:\Program Files\Java\jre1.8.0_31\lib\sunr
sasign.jar;C:\Program Files\Java\jre1.8.0_31\lib\jsse.jar;C:\Program Files\Java\
jre1.8.0_31\lib\jce.jar;C:\Program Files\Java\jre1.8.0_31\lib\charsets.jar;C:\Pr
ogram Files\Java\jre1.8.0_31\lib\jfr.jar;C:\Program Files\Java\jre1.8.0_31\class
es
java.vendor: Oracle Corporation
sun.stderr.encoding: cp437
file.separator: \
java.vendor.url.bug: http://bugreport.sun.com/bugreport/
sun.io.unicode.encoding: UnicodeLittle
sun.cpu.endian: little
sun.stdout.encoding: cp437
sun.desktop: windows
sun.cpu.isalist: amd64
```

Using the **TreeMap** Class

The `TreeMap` class executes a map with the use of a tree internally to store data. The inheritance diagram for this class is as follows:

```
java.lang.Object
  |____java.util.AbstractMap
        |____java.util.TreeMap
```

You'll find the constructors for the `TreeMap` class in Table 21.34 and its methods in Table 21.35:

Table 21.34: Constructors of the TreeMap class

Constructor	Does this
`TreeMap()`	It constructs a new, empty tree map, using natural ordering of its keys.
`TreeMap(Comparator<? super K> comparator)`	It constructs a new, empty tree map, ordered according to the given comparator.
`TreeMap(Map<? extends K,? extends V> m)`	It constructs a new tree map containing the same mappings as the given map, ordered according to the natural ordering of its keys.
`TreeMap(SortedMap<K,? extends V> m)`	It constructs a new tree map containing the same mappings and using the same ordering as the given `sorted map`.

Table 21.35: Methods of the TreeMap class	
Method	**Does this**
`Map.Entry(K,V) ceilingEntry (K key)`	It returns a key-value mapping associated with the least key greater than or equal to the given key, or null if there is no such key.
`K ceilingKey (K key)`	It returns the least key greater than or equal to the given key, or null if there is no such key.
`void clear()`	It removes all of the mappings from this map.
`Object clone()`	It gets a copy of this `TreeMap` instance.
`Comparator<? super K> comparator()`	It gets the comparator used to order the keys in this map, or null if this map uses the natural ordering.
`boolean containsKey (Object key)`	It returns True if this map contains a mapping for the specified key.
`boolean containsValue (Object value)`	It returns True if this map maps one or more keys to the specified value.
`NavigableSet<K> descendingKeySet()`	It returns a reverse order NavigableSet view of the keys contained in this map.
`NavigableMap<K,V> descendingMap()`	It returns a reverse order view of the mappings contained in this map.
`Set<Map.Entry<K,V>> entrySet()`	It returns a Set.
`Map.Entry<K,V> firstEntry()`	It returns a key-value mapping associated with the least key in this map, or null if the map is empty.
`K firstKey()`	It returns the first (lowest) key currently in this map.
`Map.Entry<K,V> floorEntry(K key)`	It returns a key-value mapping associated with the greatest key less than or equal to the given key, or null if there is no such key.
`K floorKey(Object key)`	It returns the greatest key less than or equal to the given key, or null if there is no such key.
`void forEach(BiConsumer<? super K,? super V> action)`	It performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
`V get(Object key)`	It gets the value to which the specified key is mapped, or null if this map contains no mapping for the key.
`SortedMap<K,V> headMap (K toKey)`	Equivalent to `navigableHeadMap` (Object `toKey`) `SortedMap` interface.
`Map.Entry<K,V> higherEntry(K key)`	It returns a key-value mapping associated with the least key strictly greater than the given key or null if there is no such key.
`K higherKey(K key)`	It returns the least key strictly greater than the given key or null if there is no such key.
`Set<K> keySet()`	It returns a `Set`.
`Map.Entry<K,V> lastEntry()`	It returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
`K lastKey()`	It gets the last key currently in this map.
`Map.Entry< K key, V value > lowerEntry(K key)`	It returns a key-value mapping associated with the greatest key strictly less than the given key, or null if the map is empty.
`K lowerKey(K key)`	It returns the greatest key strictly less than the given key, or null if there is no such key.
`SortedMap<K,V> headMap (K toKey)`	It returns a view of the portion of this map whose keys are strictly less than `toKey`.
`SortedMap<K,V> subMap(K formKey, K toKey)`	It returns a view of the portion of this map whose keys range from `fromKey` (inclusive) to `toKey` (exclusive).
`SortedMap<K,V> tailMap(K formKey)`	It returns a view of the portion of this map whose keys are greater than or equal to `formKey`.

Table 21.35: Methods of the TreeMap class	
Method	**Does this**
`Map.Entry<K key, V value> pollFirstEntry()`	It removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty.
`Map.Entry<K,V> pollLastEntry()`	It removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
`V put(K key, Object V)`	It associates the specified value with the specified key in this map.
`void putAll (Map<? extends K,? extends V> map)`	It copies all the mappings from the specified map to this map.
`V remove(Object key)`	It removes the mapping for this key.
`int size()`	It gets the number of key/value mappings.
`SortedMap<K,V> subMap(K fromKey, K toKey)`	Equivalent to `navigableSubMap` (Object `formKey`, Object `toKey`) `SortedMap` interface.
`SortedMap<K,V> tailMap(K fromKey)`	Equivalent to `navigableTailMap` (Object `formKey`) `SortedMap` interface.
`Collection<V> values()`	It returns a Collection.

Using the **Arrays** Class

"What are you doing?" you ask the Novice Programmer. "Writing code to sort an array," replies the NP. "Why don't you just use the `Arrays` class?" you ask. "How's that?" the NP asks.

The `Arrays` class can be used to work with arrays, including sorting and searching arrays. The inheritance diagram for this class is as follows:

```
java.lang.Object
  |____java.util.Arrays
```

You'll find the methods for the `Arrays` class in Table 21.36:

Table 21.36: Methods of the Arrays class	
Method	**Does this**
`static <T> List<T> asList(T… a)`	It gets a fixed-size list.
`static int binarySearch (byte[] a, byte key)`	It searches the given array of bytes for the given value.
`static int binarySearch (char[] a, char key)`	It searches the given array of characters for the given value.
`static int binarySearch (double[] a, double key)`	It searches the given array of `doubles` for the given value.
`static int binarySearch (float[ ] a, float key)`	It searches the given array of `floats` for the given value.
`static int binarySearch (int[] a, int key)`	It searches the given array of `ints` for the given value.
`static int binarySearch (long[] a, long key)`	It searches the given array of `longs` for the given value.
`static int binarySearch (Object[] a, Object key)`	It searches the given array for the given object.
`static <T> int binarySearch(T[] a, T key, Comparator<? super T> c)`	It searches the given array for the given object using a comparator.
`static int binarySearch (short[] a, short key)`	It searches the given array of `shorts` for the given value.

Table 21.36: Methods of the Arrays class

Method	Does this
`static boolean[] copyOf (boolean[] original, int newLength)`	It copies the specified array, truncating or padding with false (if necessary) so the copy has the specified length.
`static byte[] copyOf(byte[] original, int newLength)`	It copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
`static char[] copyOf(char[] original, int newLength)`	It copies the specified array, truncating or padding with null characters (if necessary) so the copy has the specified length.
`static double[] copyOf (double[] original, int newLength)`	It copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
`static float[] copyOf (float[] original, int newLength)`	It copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
`static int[] copyOf(int[] original, int newLength)`	It copies the specified array, truncating or padding with zeros(if necessary) so the copy has the specified length.
`static long[] copyOf(long[] original, int newLength)`	It copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
`static short[] copyOf (short[] original, int newLength)`	It copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
`static boolean equals (boolean[] a, boolean[] a2)`	It returns True if the two given arrays of `booleans` are equal.
`static boolean equals (byte[] a, byte[] a2)`	It returns True if the two given arrays of `bytes` are equal to one another.
`static boolean equals (char[] a, char[] a2)`	It returns True if the two given arrays of characters are equal to one another.
`static boolean equals (double[] a, double[] a2)`	It returns True if the two given arrays of `doubles` are equal.
`static boolean equals (float[] a, float[] a2)`	It returns True if the two given arrays of `floats` are equal.
`static boolean equals (int[] a, int[] a2)`	It returns True if the two given arrays of `ints` are equal.
`static boolean equals (long[] a, long[] a2)`	It returns True if the two given arrays of `longs` are equal.
`static boolean equals (Object[ ] a, Object[ ] a2)`	It returns True if the two given arrays of objects are equal.
`static boolean equals (short[ ] a, short[ ] a2)`	It returns True if the two given arrays of `shorts` are equal to one another.
`static void fill (boolean[] a, boolean val)`	It assigns the given `boolean` value to each element of the given array of `Booleans`.
`static void fill(boolean[] a, int fromIndex, int toIndex, boolean val)`	It assigns the given `boolean` value to each element of the given range of the given array of `Booleans`.
`static void fill(byte[] a, byte val)`	It assigns the given `byte` value to each element of the given array of `bytes`.
`static void fill(byte[] a, int fromIndex, int toIndex, byte val)`	It assigns the given `byte` value to each element of the given range of the given array of `bytes`.

Table 21.36: Methods of the Arrays class

Method	Does this
`static void fill(char[] a, char val)`	It assigns the given `character` value to each element of the given array of `characters`.
`static void fill(char[] a, int fromIndex, int toIndex, char val)`	It assigns the given character value to each element of the given range of the given array of characters.
`static void fill(double[] a, double val)`	It assigns the given `double` value to each element of the given array of `doubles`.
`static void fill(double[] a, int fromIndex, int toIndex, double val)`	It assigns the given `double` value to each element of the given range of the given array of `doubles`.
`static void fill(float[] a, float val)`	It assigns the given `float` value to each element of the given array of `floats`.
`static void fill(float[] a, int fromIndex, int toIndex, float val)`	It assigns the given `float` value to each element of the given range of the given array of `floats`
`static void fill(int[] a, int val)`	It assigns the given `int` value to each element of the given array of `ints`.
`static void fill(int[] a, int fromIndex, int toIndex, int val)`	It assigns the given `int` value to each element of the given range of the given array of `ints`.
`static void fill(long[] a, int fromIndex, int toIndex, long val)`	It assigns the given `long` value to each element of the given range of the given array of `longs`.
`static void fill(long[] a, long val)`	It assigns the given `long` value to each element of the given array of `longs`
`static void fill(Object[] a, int fromIndex, int toIndex, Object val)`	It assigns the given `object` reference to each element of the given range of the given array of `objects`.
`static void fill(Object[] a, Object val)`	It assigns the given `object` reference to each element of the given array of `objects`.
`static void fill(short[] a, int fromIndex, int toIndex, short val)`	It assigns the given `short` value to each element of the given range of the given array of `shorts`.
`static void fill(short[] a, short val)`	It assigns the given `short` value to each element of the given array of `shorts`.
`static void sort(byte[] a)`	It sorts the given array of `bytes` into ascending numerical order.
`static void sort(byte[] a, int fromIndex, int toIndex)`	It sorts the given range of the given array of bytes into ascending numerical order.
`static void sort(char[] a)`	It sorts the given array of characters into ascending numerical order.
`static void sort(char[] a, int fromIndex, int toIndex)`	It sorts the given range of the given array of characters into ascending numerical order.
`static void sort (double[] a)`	It sorts the given array of `doubles` into ascending numerical order.
`static void sort(double[] a, int fromIndex, int toIndex)`	It sorts the given range of the given array of `doubles` into ascending numerical order.
`static void sort (float[] a)`	It sorts the given array of `floats` into ascending numerical order.
`static void sort(float[] a, int fromIndex, int toIndex)`	It sorts the given range of the given array of `floats` into ascending numerical order.

Table 21.36: Methods of the Arrays class	
Method	**Does this**
`static void sort(int[] a)`	It sorts the given array of `ints` into ascending numerical order.
`static void sort(int[] a, int fromIndex, int toIndex)`	It sorts the given range of the given array of `ints` into ascending numerical order.
`static void sort(long[] a)`	It sorts the given array of `longs` into ascending numerical order.
`static void sort(long[] a, int fromIndex, int toIndex)`	It sorts the given range of the given array of `longs` into ascending numerical order.
`static void sort (Object[] a)`	It sorts the given array of `objects` into ascending order, according to the natural ordering of its elements.
`static void sort(Object[] a, Comparator c)`	It sorts the given array of `objects` according to the order induced by the given comparator.
`static void sort(Object[] a, int fromIndex, int toIndex)`	It sorts the given range of the given array of `objects` into ascending order, according to the natural ordering of its elements.
`static void sort(Object[] a, int fromIndex, int toIndex, Comparator c)`	It sorts the given range of the given array of `objects` according to the order induced by the given comparator.
`static void sort(short[] a)`	It sorts the given array of `shorts` into ascending numerical order.
`static void sort(short[] a, int fromIndex, int toIndex)`	It sorts the given range of the given array of `shorts` into ascending numerical order.

Here's an example in which we create an array, print it out, sort it, and then search for a particular element:

```
import java.util.*;

class ArrayDemo
{
  public static void main(String args[])
  {
        int array[] = new int[10];
        for(int loop_index = 9; loop_index > 0; loop_index--)
        array[loop_index] = -loop_index;
        for(int loop_index = 0; loop_index < array.length; loop_index++)
        System.out.print(array[loop_index] + " ");
        System.out.println();

        Arrays.sort(array);
        for(int loop_index = 0; loop_index < array.length; loop_index++)
        System.out.print(array[loop_index] + " ");
        System.out.println();
        System.out.print("Found -5 at position " +
        Arrays.binarySearch(array, -5));
  }
}
```

Note that these are static methods and can be used without instantiating an object of this class.

Here's the output of this example:

```
C:\>java ArrayDemo
0 -1 -2 -3 -4 -5 -6 -7 -8 -9
-9 -8 -7 -6 -5 -4 -3 -2 -1 0
Found -5 at position 4
```

Learning the Fundamentals of Enumerations

The Novice Programmer appears in a serious mood. Puffing on his cigar, he says, "If we talk about any other computer language, C++ for example, there is a term known as enumeration. As named integer constants, these

features help the user in a great way. But what about Java?" You say, "No, you are wrong. The long felt need is over. Now, there is an enumeration feature in Java too. But it functions a little different from other codes."

Let's talk about the basics of enumerations in detail. In the original version of Java, one feature is absent and many programmers felt that this feature is one of the vital features in a code.

Many programmers felt that this was needed very much. This is known as Enumeration. In its simplest form, you can define Enumeration as a list of named constants.

An Enumeration type can be considered to be a special form of class while defining an enumeration type at the time of writing code. The specified enumeration constant gets created as objects of the class that contains the `enum` class. This class is defined in the `java.lang` package as a superclass. The object corresponding to each enumeration constants stores the constant's name in a field. The `toString()` method is inherited by the enumeration class type from the `enum` class.

Enumeration in Java is different in existence from other languages, though it seems to be similar. In C++, enumeration is a list of named integer constants, while in Java, enumeration defines a class type. This concept is elaborated by making enumeration into classes.

Using the new `enum` keywords, an Enumeration is created. For example, given here is a simple enumeration that lists various Wheat varieties:

```
enum Wheat { Durum, Spelt, CommonWheat, Emmer }
```

Enumeration constants are the identifiers such as Durum, Spelt, and so on each of which is implicitly declared as a `public` and `static` member of Wheat. In addition to this, their type is the type of enumeration in which they are declared, which is Wheat in this case. These constants in Java are known as self-typed which self corresponds to the enclosed enumeration.

You can create a variable of enumeration type once you have defined enumeration. However, it is found that, even though enumeration defines a class type, you do not instantiate an `enum` by using a new. Instead, you declare and use an enumeration variable in much the same way as you have done in the case of the primitive types. For example, the code given here declares al as a variable of enumeration type Wheat:

```
Wheat al ;
```

The only values that `al` can assign are those defined by the enumeration, because `al` is of type Wheat. For example, the value Emmer is assigned to `al` by this:

```
al= Wheat.Emmer;
```

The point to note here is that the symbol Emmer is preceded by Wheat. The == relational operator can be used for comparing the two enumeration constants for equality.

For example, this statement compares the value in `al` with the Spelt constant:

```
if(al = = Wheat.Spelt) //….
```

An enumeration value can also be used for controlling a switch statement. Actually, it is very necessary that all of the `case` statements must use constants from the same `enum` as that used by the `switch` expression. For example, this `switch` is perfectly valid:

```
switch (al) {
  case Spelt:
  case CommonWheat :
```

In the `case` statements, you can note that the names of the enumeration constants can be used without being qualified by the name of the enumeration type. For example, `CommonWheat` is used, but not `Wheat.CommonWheat`. The reason behind this is that the type of the enumeration in the `switch` expression has already implicitly declared the `enum` type of the case constants. So, for `case` statement, there is no requirement for the constants to be passed with their `enum` type name. In fact, a compilation error will occur if you will do so. When an enumeration constant is shown, such as in a `println()` statement, the result will be its name. For example, in the given statement:

```
System.out.println(Wheat.Durum);
```

The name Durum is displayed as output. The code explained here will put together all the pieces and examine the Wheat enumeration:

```
enum Wheat { Emmer,Durum, CommonWheat, Spelt }
class EnumDemo
```

```
{
  public static void main (String args[])
  {
        Wheat al;
        al = Wheat.Emmer;
        System.out.println("Value of al is : " + al);
        System.out.println();
        al = Wheat.Emmer;
        if (al == Wheat.Spelt)
        {
                System.out.println ("al contains Spelt.\n");
        }
        switch(al) {
        case Emmer:
                System.out.println("It is ancient time wheat."); break;
        case Durum:
                System.out.println("It is second most widely cultivated wheat.");
                break;
                case CommonWheat:
                System.out.println("It is most wildly cultivated wheat."); break;
        case Spelt:
                System.out.println("It is found in limited quantities."); break;
        }
  }
}
```

Here's the output is as follows:

```
Value of al is: Emmer
It is ancient time wheat.
```

The *values()* and *valueOf()* Methods

Two predefined methods, namely, `values()` and `valueOf()`, are present automatically in all enumeration. Given are their general forms. One form is:

```
public static enum-type[] values()
```

And the other is:

```
public static enum-type valueOf(String str)
```

The `values()` method returns an array containing a set of the enumeration constants. On the other hand, the enumeration constant whose value corresponds to the string passed in `str` is returned by the `valueOf()` method. In both cases it is found that enum-type is the type of the enumeration. For example, in the case of the `Wheat` enumeration shown earlier, the return type of `Wheat.valueOf("Durum")` is `Durum`. The `values()` and `valueOf()` methods are demonstrated by the following code:

```
enum Wheat { Emmer,Durum, CommonWheat, Spelt  }
class WheatC
{
  public static void main (String args[])
  {
        Wheat al;
        System.out.println("Here are all Wheat constant:");
        Wheat allwheat[] = Wheat.values();
        for(wheat a : allwheat)
        System.out.println(a);
        System.out.println();
        al = Wheat.valueOf("Durum");
        System.out.println("al contains "+al);
  }
}
```

Here's the output of the code as follows:

```
Here are all Wheat constants:
Emmer
Durum
CommonWheat
```

```
Spelt

al contains Durum
```

The point to be noted here is that this code uses a `for-each` style for loop to cycle through the array of constants which can be obtained by calling `values()`. For better understanding, the variable `allwheat` was formed and thus assigned a reference to the enumeration array. However, this step is not necessary to explain. Because, by another way, the `for` could have been written as shown here by eliminating the need for the `allwheat` variable:

```
for (Wheat a : Wheat.values()) { System.out.println(a); }
```

Now, you have to notice that, by calling `valueOf()`, the value corresponding to the name Durum is obtained:

```
al = Wheat.ValueOf ("Durum");
```

We have discussed in the preceding topic that `valueOf()` returns the enumeration value related to the name of the constant which is denoted as a string.

Java Enumeration as a Class Type

As already explained, the enumeration of Java is a class type. Although you do not instantiate an `enum` using new, it has much more the same capabilities as other classes have. Here, the important fact is that `enum` defines a class that gives power to the Java enumeration. However, as far as other codes are concerned, that enumeration simply does not have this priority. For example, you can give them constructors, add instance variables and methods, and even implement interfaces.

You must keep in mind that each enumeration constant is an instance of its own enumeration type. Thus, at the time when you define a constructor for an `enum`, the constructor is called only after each enumeration constant is created. Moreover, each enumeration constant contains its own copy of an instance variable, which is defined by the enumeration. For example, see the following version of the `Mango`:

```
enum Mango {
  Brooks(10), Manilla(9), Alphanso(12), Kesar(15), Maya(8);
  private int price;
  Mango(int p) { price = p;}
  int getPrice() { return price;}
}
class Apple5
{
  public static void main(String args[])
  {
        Mango ap;
        System.out.println("Kesar costs "+Mango.Kesar.getPrice()+ " cents. \n");
        System.out.println("All mango prices:");
        for(Mango a : Mango.values())
        System.out.println(a + " costs " + a.getPrice() + " cents.");
  }
}
```

The output of the code is as follows:

```
Kesar costs 15 cents.

All mango prices:
Brooks costs 10 cents.
Manilla costs 9 cents.
Alphanso costs 12 cents.
Kesar costs 15 cents.
Maya costs 8 cents.
```

Here, the version of `Mango` adds three things to it. The first one which is used to hold the price of each variety of `Mango` is the instance variable `price`. The second one is the `Mango` constructor, which is to pass the price of a `Mango`. And finally, the third one which returns the value of price is the method `getPrice()`.

When the variable ap is declared in main(), then the constructor for Mango is called once for each of the constant which is specified. You have to notice how the argument to the constructor is specified, by putting them inside parentheses after each constant, as shown here:

```
Brooks(10), Manilla(9), Alphanso(12), Kesar(15), Maya(8);
```

Here, these values are passed to the p parameter of Mango(), and after that, it assigns this value to price. Again, for each of the constant, the constructor is called only once. The reason being that each enumeration constant has its own copy of price. In order to obtain the price of a specified type of Mango, you can have it by calling getPrice(). For example, in main(), the price of a Kesar is obtained by the following call:

```
Mango.Kesar.getPrice()
```

In order to obtain the prices of all varieties, you have to go by cycling through the enumeration using a for loop. The reason is, there is a copy of price for each enumeration constant, and also the value associated with one constant is separate and distinct from the value associated with another constant. As you have already noticed, this is a powerful concept. This concept is only available when enumeration is implemented as classes, as Java does.

In the preceding example, there is only one constructor, but an enum can provide multiple overloaded forms, like it provides in any other class. For example, this version of Mango provides a default constructor that initializes the price to –1, just to indicate that there is no price data available here:

```
enum Mango
{
  Brooks(10), Manilla(9), Alphanso(), Kesar(15), Maya(8);
  private int price;
  Mango(int p) { price = p; }
  Mango() {price = -1; }
  int getPrice() { return price; }
}
```

However, please note that in this version, Alphanso is not given as an argument. This means that the default constructor is called and the value –1 has been given to Alphanso's price variable.

Let's mention about two different restrictions that apply to the enumeration. The first restriction to remember is that an enumeration can't inherit another class. The second restriction is, an enum cannot be a superclass. To make it simpler, an enum can't be extended.

Enumeration Inheriting Enum

We know that at declaration time for enum, you can't inherit a superclass. Therefore, all enumerations automatically inherit one: java.lang.Enum. This class defines various methods that are meant to be used by all enumerations. The constructor and methods are shown in Table 21.37 and Table 21.38, respectively:

Table 21.37: Showing the constructor of the Enum

Constructor	Does this
protected Enum(String name, int ordinal)	The only constructor

Table 21.38: Showing the methods of the Enum

Methods	Does this
protected Object clone()	It throws CloneNotSupportedException.
int compareTo(E o)	It compares this enum with the specified object for order.
boolean equals (Object other)	It returns true if the specified object is equal to this enum constant.
protected void finalize()	Enum classes cannot have finalized methods.
Class<E> getDeclaringClass()	It returns the Class object corresponding to this enum constant's enum type.
int hashCode()	It returns a hash code for this enum constant.
String name()	It returns the name of this enum constant, exactly as declared in its enum declaration.

Table 21.38: Showing the methods of the Enum	
Methods	**Does this**
`int ordinal()`	It returns the ordinal of this enumeration constant (its position in its `enum` declaration, where the initial constant is assigned an ordinal of zero).
`String toString()`	It returns the name of this `enum` constant, as contained in the declaration.
`static valueOf(Class<T> enumType, String name)`	It returns the `enum` constant of the specified `enum` type with the specified name`.

Questions that arise over here are what do you mean by ordinal value and how can you achieve it? You can obtain a value that symbolizes an enumeration constant's position in the list of constants. This is obtained by calling the `ordinal()` method as follows:

```
final int ordinal()
```

The preceding statement returns the ordinal value of the invoking constant. Note that ordinal values begin at Zero. Thus, `Brooks` has an ordinal value of Zero, `Manilla` has an ordinal value of 1, `Alphanso` has an ordinal value of 2, and so on and so forth in the `Mango` enumeration.

You can perform comparison between the ordinal values of two constants of the similar enumeration by using the `compareTo()` method. It has this general form:

```
final int compareTo(enum-type e)
```

where `enum`-type is the type of enumeration and `e` is the constant that is compared to the constant being requested. Also keep in mind that both the requested constant and `e` must be of the same enumeration. If the invoking constant has an ordinal value less than e's, then `compareTo()` returns a negative value. Moreover, if the two ordinal values are similar, then zero is returned, and if the value of the invoking constant is greater than e's, then `compareTo()` returns a positive value.

To obtain the equality, you can compare an enumeration constant with any other object by using `equals()`, which overrides the `equal()` method defined by `Object`. Although `equals()` can compare an enumeration constant to any other object, those two objects will remain equal if both of them refer to the same constant, that too within the same enumeration. For better understanding, make a note that having ordinal values in common will not cause `equals()` to return true if the two constants are from different enumerations.

As discussed earlier, you can perform the comparison between two enumeration references for equality by using the == operator. You will get it by the following code which examines the `ordinal()`,`compareTo()`, and `equals()` methods:

```
enum Mango { Brooks, Manilla, Alphanso, Kesar, Maya }
class Apple6
{
  public static void main (String args[])
  {
        Mango ap, ap2, ap3;
        System.out.println("Here are all mango constants" + " and their
        ordinal values: ");
        for (Mango a : Mango.values())
                System.out.println(a +" " + a.ordinal());
        ap = Mango.Alphanso;
        ap2 = Mango.Manilla;
        ap3 = Mango.Alphanso;
        System.out.println();
        if(ap.compareTo(ap2) < 0)
                System.out.println(ap + " comes before " + ap2);
        if(ap.compareTo(ap2) > 0)
                System.out.println(ap2 + " comes before " + ap);
        if(ap.compareTo(ap2) == 0)
                System.out.println(ap + " equals " + ap3);
        System.out.println();
        if(ap.equals(ap2)) { System.out.println("Error"); }
        if(ap.equals(ap3)) { System.out.println("Error"); }
```

```
            if(ap == ap3) { System.out.println(ap + " = = " + ap3); }
        }
    }
```

Here's the output of the code:

```
Here are all mango constants and their ordinal values:
Brooks 0
Manilla 1
Alphanso 2
Kesar 3
Maya 4

Manilla comes before Alphanso
Error
Alphanso = = Alphanso
```

The **Enumeration** Interface

You use the `Enumeration` interface's methods to loop over elements in classes such as `Hashtable`. This interface is a useful one, and we'll put it to work later in this chapter. For now, we'll list its methods for reference. You'll find the methods for the `Enumeration` interface in Table 21.39:

Table 21.39: Methods of the Enumeration interface

Method	Does this
`boolean hasMoreElements()`	It returns True if this enumeration contains more elements.
`Object nextElement()`	It gets the next element of this enumeration.

The **Legacy** Classes and Interfaces

Talking about earlier versions of Java, legacy classes and interfaces formed the collections framework and now they are reconstructed in coming versions. With the Java SE 7 release, for legacy file I/O code, the `java.io.File` class was the process used for file I/O; however, it has the following demerits:

- Many methods didn't throw exceptions when they failed, so it was impossible to retrieve a useful error message.
- The rename method didn't work properly across platforms.
- There was no real support for symbolic links.
- Needed more support for metadata, such as file permissions, file owner, and other security attributes.
- Executing file metadata was inefficient.
- It was impossible to write a reliable code that could iteratively walk a file tree and result back accurately if there were circular symbolic links.

So, you have the legacy code that uses `java.io.File` and would prefer taking benefit of the `java.nio.file.Path` functionality with minimal impact on your code.

The various legacy classes defined by `java.util` package are:

- Dictionary
- Hashtable
- Stack
- Vector
- Properties

The *Java Native Interface* is developed for using Java in order to encapsulate native legacy code on small embedded platforms. We must know why existing technologies for encapsulating legacy code, Java Native Interface (JNI), is not enough for an important range of small embedded platforms, and we depict how the JNI provides previously missing functionality.

The *Vector* Class

The `Vector` class predates the collection framework, and it implements a dynamic array. Because of the appearance of the collection framework, `Vector` has been rewritten to be compatible with it. Here's the inheritance diagram for `Vector`:

```
java.lang.Object
  |____java.util.AbstractCollection
        |____java.util.AbstractList
              |____java.util.Vector
```

You'll find the fields of the `Vector` class in Table 21.40, its constructors in Table 21.41, and its methods in Table 21.42:

Table 21.40: Fields of the Vector class

Field	Does this
`protected int capacityIncrement`	This is the amount by which the capacity of the vector is increased when its size exceeds its capacity.
`protected int elementCount`	Total number of components in this `Vector` object.
`protected Object[] elementData`	The array where the components of the vector are stored.

Table 21.41: Constructors of the Vector class

Constructor	Does this
`Vector()`	It constructs an empty vector.
`Vector(Collection c)`	It constructs a vector containing the elements of the given collection.
`Vector(int initialCapacity)`	It constructs an empty vector with the given initial capacity.
`Vector(int initialCapacity, int capacityIncrement)`	It constructs an empty vector with the given initial capacity and capacity increment.

Table 21.42: Methods of the Vector class

Method	Does this
`void add(int index, Object element)`	It inserts the given element.
`boolean add(Object o)`	It adds the given element to the end of this vector.
`boolean addAll (Collection c)`	It adds all the elements in the given collection to the end of this vector in the order that they are returned by the given collection's iterator.
`boolean addAll(int index, Collection c)`	It inserts all the elements in the given collection into this vector at the given position.
`void addElement(Object obj)`	It adds the given component to the end of this vector, increasing its size by one.
`int capacity()`	It gets the current capacity of this vector.
`void clear()`	It removes all the elements from this vector.
`Object clone()`	It gets a clone of this vector.
`boolean contains (Object elem)`	It returns True if this vector contains the specified element.
`boolean containsAll (Collection c)`	It returns True if this vector contains all the elements in the given collection.
`void copyInto(Object[] anArray)`	It copies the components of this vector into the given array.
`Object elementAt(int index)`	It gets the component at the given index.
`Enumeration elements()`	It gets an enumeration of the components of this vector.
`void ensureCapacity (int minCapacity)`	It increases the capacity of this vector, if necessary.

Table 21.42: Methods of the Vector class	
Method	**Does this**
`boolean equals(Object o)`	It compares the given object with this vector for equality.
`Object firstElement()`	It gets the first component of this vector.
`Object get(int index)`	It gets the element at the given position in this vector.
`int hashCode()`	It gets the hashcode value for this vector.
`int indexOf(Object elem)`	It returns the index of the first occurrence of the given element. In this vector, or -1 if this vector does not contain element.
`int indexOf (Object elem, int index)`	It searches for the first occurrence of the given argument, beginning the search at `index`.
`void insertElementAt (Object obj, int index)`	It inserts the given object as a component in this vector at the given index.
`boolean isEmpty()`	It tests whether this vector has no components.
`Object lastElement()`	It gets the last component of the vector.
`int lastIndexOf (Object elem)`	It gets the index of the last occurrence of the specified element in this vector or -1 if this vector does not contain the element.
`int lastIndexOf (Object elem)`	It returns the index of the last occurrence of the specified element in this vector, searching backwards from index, or returns -1 if the element is not found.
`int lastIndexOf (Object elem, int index)`	It searches backwards for the given object and returns an index to it.
`Object remove(int index)`	It removes the element at the given position in this vector.
`boolean remove(Object o)`	It removes the first occurrence of the given element.
`boolean removeAll (Collection c)`	It removes from this vector all its elements that are contained in the given collection.
`void removeAllElements()`	It removes all components from this vector and sets its `size` to zero.
`boolean removeElement(Object obj)`	It removes the first occurrence of the argument from this vector.
`void removeElementAt (int index)`	It deletes the component at the given index.
`protected void removeRange (int fromIndex, int toIndex)`	It removes from this list all the elements whose indexes are between `fromIndex` and `toIndex`.
`boolean retainAll (Collection c)`	It keeps only the elements in this vector that are contained in the given collection.
`Object set (int index, Object element)`	It replaces the element at the given position in this vector with the given element.
`void setElementAt (Object obj, int index)`	It sets the component at the given index of this vector to be the given object.
`void setSize(int newSize)`	It sets the size of this vector.
`int size()`	It gets the number of components in this vector.
`List subList(int fromIndex, int toIndex)`	It gets a view of the portion of this list between `fromIndex` and `toIndex`.
`Object[] toArray()`	It gets an array containing all the elements in this vector.
`Object[] toArray (Object[] a)`	It gets an array containing all the elements in this vector.
`String toString()`	It gets a string representation of this vector.
`void trimToSize()`	It trims the capacity of this vector to the current size.

Each vector has a *capacity*, which is the maximum size it can grow to. When you exceed that size, the capacity is automatically increased. You can use the `add()` and `addElement()` methods to add elements to a vector, the `remove()` and `removeElement()` methods to remove elements, and the `contains()` method to do a search. Here's an example using the `Vector` class in which we create a vector, check its size and capacity, and search for an element:

```
import java.util.*;

class VectorDemo
{
  public static void main(String args[])
  {
        Vector<Number>  vector = new Vector<Number>(5);
        System.out.println("Capacity: " + vector.capacity());
        vector.addElement(new Integer(0));
        vector.addElement(new Integer(1));
        vector.addElement(new Integer(2));
        vector.addElement(new Integer(3));
        vector.addElement(new Integer(4));
        vector.addElement(new Integer(5));
        vector.addElement(new Integer(6));
        vector.addElement(new Double(3.14159));
        vector.addElement(new Float(3.14159));
        System.out.println("Capacity: " + vector.capacity());
        System.out.println("Size: " + vector.size());
        System.out.println("First item: " + (Integer)
        vector.firstElement());
        System.out.println("Last item: " + (Float) vector.lastElement());
        if(vector.contains(new Integer(3)))
        System.out.println("Found a 3.");
  }
}
```

Here's the output of this example:

```
C:\>java VectorDemo
Capacity: 5
Capacity: 10
Size: 9
First item: 0
Last item: 3.14159
Found a 3.
```

The *Stack* Class

The Novice Programmer is back and says, "I wish I could reverse the elements of a list—I'm converting a number to base 20, and each time I divide by 20, the new digits come off in reverse order." "No problems," you say. "Just put them on a stack and pop them to reverse the order." "Great!" says the NP.

The `Stack` class is built on the `Vector` class, and it implements a stack construct. Here's the inheritance diagram for the `Stack` class:

```
java.lang.Object
  |____java.util.AbstractCollection
        |____java.util.AbstractList
              |____java.util.Vector
                    |____java.util.Stack
```

You'll find the constructor for the `Stack` class in Table 21.43 and its methods in Table 21.44:

Table 21.43: The constructor of the Stack class

Constructor	Does this
`Stack()`	It creates an empty stack.

Table 21.44: Methods of the Stack class	
Method	**Does this**
`boolean empty()`	It returns True if this stack is empty.
`Object peek()`	It gets the object at the top of this stack without removing it.
`Object pop()`	It removes the object at the top of this stack and returns that object.
`Object push(Object item)`	It adds an item onto the top of this stack.
`int search(Object o)`	It gets the position of an object on this stack.

You use the `push()` method to add an element to a stack and the `pop()` method to retrieve it; note that elements come off a stack in the reverse order in which they were added to it. Here's an example in which we push and then pop elements on a stack:

```
import java.util.*;

class StackDemo
{
  public static void main(String args[])
  {
        Stack<Integer> stack1 = new Stack<Integer>();
        try {
                stack1.push(new Integer(0));
                stack1.push(new Integer(1));
                stack1.push(new Integer(2));
                stack1.push(new Integer(3));
                stack1.push(new Integer(4));
                stack1.push(new Integer(5));
                stack1.push(new Integer(6));

                System.out.println((Integer) stack1.pop());
                System.out.println((Integer) stack1.pop());
                System.out.println((Integer) stack1.pop());
                System.out.println((Integer) stack1.pop());
                System.out.println((Integer) stack1.pop());
                System.out.println((Integer) stack1.pop());
                System.out.println((Integer) stack1.pop());
        }
                catch (EmptyStackException e) { }
  }
}
```

Here's the output of this example:

```
C:\>java StackDemo
6
5
4
3
2
1
0
```

The *Dictionary* Class

The `Dictionary` class is a class used by classes such as `Hashtable`, and it stores elements much like a map does, although it was introduced before the collection framework appeared. Because it forms the foundation of the `Hashtable` class, we'll include `Dictionary` here for reference. Here's the inheritance diagram for `Dictionary`:

```
java.lang.Object
  |____java.util.Dictionary
```

You'll find the constructor for `Dictionary` in Table 21.45 and its methods in Table 21.46:

Table 21.45: The constructor of the Dictionary class

Constructor	Does this
`Dictionary()`	It creates a dictionary.

Table 21.46: Methods of the Dictionary class

Method	Does this
`abstract Enumeration elements()`	It gets an enumeration of the values in this dictionary.
`abstract Object get (Object key)`	It gets the value for this key.
`abstract boolean isEmpty()`	It returns True if this dictionary has no keys.
`abstract Enumeration keys()`	It gets an enumeration of the keys in this dictionary.
`abstract Object put (Object key, Object value)`	It maps the given key to the given value in this dictionary.
`abstract Object remove (Object key)`	It removes the key and its value from this dictionary.
`abstract int size()`	It gets the number of entries in this dictionary.

The *Hashtable* Class

The `Hashtable` class implements the `Map` interface, and it was how you used to implement maps before the collection framework come. Here's the inheritance diagram for `Hashtable`:

```
java.lang.Object
  |____java.util.Dictionary
        |____java.util.Hashtable
```

You'll find the constructors for the `Hashtable` class in Table 21.47 and its methods in Table 21.48:

Table 21.47: Constructors of the Hashtable class

Constructor	Does this
`Hashtable()`	It constructs a new, empty hashtable.
`Hashtable (int initialCapacity)`	It constructs a new, empty hashtable with the given initial capacity.
`Hashtable(int initialCapacity, float loadFactor)`	It constructs a new, empty hashtable with the given initial capacity and the given load factor.
`Hashtable(Map t)`	It constructs a new hashtable with the same mappings as the given map.

Table 21.48: Methods of the Hashtable class

Method	Does this
`void clear()`	It clears this hashtable so that it contains no keys.
`Object clone()`	It creates a copy of this hashtable.
`boolean contains (Object value)`	It returns True if some key maps into the given value in this hashtable.
`boolean containsKey (Object key)`	It returns True if the given object is a key in this hashtable.
`boolean containsValue (Object value)`	It returns True if this hashtable maps one or more keys to this value.
`Enumeration elements()`	It gets an enumeration of the values in this hashtable.
`Set entrySet()`	It gets a set view of the entries contained in this hashtable.
`boolean equals(Object o)`	It compares the given object with this map for equality.
`Object get(Object key)`	It gets the value to which the given key is mapped in this hashtable.

Table 21.48: Methods of the Hashtable class	
Method	**Does this**
`int hashCode()`	It gets the hashcode value for this map, as per the definition in the `Map` interface.
`boolean isEmpty()`	It tests whether this hashtable maps no keys to values.
`Enumeration keys()`	It gets an enumeration of the keys in this hashtable.
`Set keySet()`	It gets a set view of the keys contained in this hashtable.
`Object put (Object key, Object value)`	It maps the given key to the given value in this hashtable.
`void putAll(Map t)`	It copies all the mappings from the given map to this hashtable.
`protected void rehash()`	It increases the capacity of and internally reorganizes this hashtable.
`Object remove(Object key)`	It removes the key (and its corresponding value) from this hashtable.
`int size()`	It gets the number of keys in this hashtable.
`String toString()`	It gets a string representation of this `Hashtable` object in the form of a set of entries.
`Collection values()`	It gets a collection view of the values contained in this hashtable.

You store and retrieve key/value pairs in a hashtable using the `put()` and `get()` methods. You can get an `Enumeration` object for the keys in a hashtable using the `keys()` method, and you can use that object to loop over the elements of the hash. Here's an example in which we create a hashtable and then print out its key/value pairs:

```
import java.util.*;

class HashTable
{
  public static void main(String args[])
  {
        Hashtable<String, String> hashtable1 = new Hashtable<String, String>();
        hashtable1.put("Item 0", "Value 0");
        hashtable1.put("Item 1", "Value 1");
        hashtable1.put("Item 2", "Value 2");
        hashtable1.put("Item 3", "Value 3");
        hashtable1.put("Item 4", "Value 4");
        hashtable1.put("Item 5", "Value 5");
        hashtable1.put("Item 6", "Value 6");
        Enumeration keys = hashtable1.keys();
        while(keys.hasMoreElements())
        {
                String key = (String) keys.nextElement();
                System.out.println(key + "/" + hashtable1.get(key));
        }
  }
}
```

Here's the output of this example. Note that the hash determines the order in which elements are stored in a hash, and you can't count on any specific order:

```
C:\>java HashTable
Item 0/Value 0
Item 6/Value 6
Item 5/Value 5
Item 4/Value 4
Item 3/Value 3
Item 2/Value 2
Item 1/Value 1
```

The **Properties** Class

The `Properties` class is derived from the `Hashtable` class (see the previous solution) and uses string keys and values. This class maintains the properties of various other classes, and an object of the `Properties` class is returned by those classes's `getProperty()` method. Here's the inheritance diagram for the `Properties` class:

```
java.lang.Object
  |____java.util.Dictionary
        |____java.util.Hashtable
              |____java.util.Properties
```

You'll find the single field of the `Properties` class in Table 21.49, its constructors in Table 21.50, and its methods in Table 21.51:

Table 21.49: The field of the Properties class

Field	Does this
`protected Properties defaults`	A property list that contains default values for keys

Table 21.50: Constructors of the Properties class

Constructor	Does this
`Properties()`	It constructs a property list.
`Properties(Properties defaults)`	It constructs a property list with the given defaults.

Table 21.51: Methods of the Properties class

Method	Does this
`String getProperty (String key)`	It searches for the property with the given key.
`String getProperty (String key, String defaultValue)`	It searches for the property with the given key and returns the default value if the key is not found.
`void list(PrintStream out)`	It prints this property list to the given output stream.
`void list(PrintWriter out)`	It prints this property list to the given writer stream.
`void load (InputStream inStream)`	It reads a property list from the input stream.
`Enumeration propertyNames()`	It gets an enumeration of all the keys in this property list.
`void save(OutputStream out, String header)`	It is deprecated and it uses the `store` method instead.
`Object setProperty (String key, String value)`	It calls the hashtable method `put`.
`void store (OutputStream out, String header)`	It writes this property list in this `Properties` table to the output stream in a format acceptable for use with the `load` method.
`Set <String> stringPropertyNames()`	It returns a set of keys in this property list where the key and its corresponding value are strings, including distinct keys in the default property list if a key of the same name has not already been found from the main properties list.

You can put property values into a `Properties` object with the `setProperty()` method, and you can get property values with the `getProperty()` method. Here's an example in which we create a `Properties` object, add property values to it, and then retrieve a property value. Note, in particular, that you can supply a default value to `getProperty()` that it will return if there's no property value matching the property you have requested:

```
import java.util.*;

class PropertiesDemo
```

```
{
  public static void main(String args[])
  {
        Properties properties1 = new Properties();
        Set states; String outString;
        properties1.setProperty("Property 0", "Value 0");
        properties1.setProperty("Property 1", "Value 1");
        properties1.setProperty("Property 2", "Value 2");
        properties1.setProperty("Property 3", "Value 3");
        properties1.setProperty("Property 4", "Value 4");
        properties1.setProperty("Property 5", "Value 5");
        properties1.setProperty("Property 6", "Value 6");
        outString = properties1.getProperty("Property 3", "Missing");
        System.out.println(outString);
        outString = properties1.getProperty("Property 7", "Missing");
        System.out.println(outString);
  }
}
```

Here's the output of this example:

```
C:\>java PropertiesDemo
Value 3
Missing
```

Using the Aggregate Operations

It has been discussed earlier that aggregate operations are performed on a data structure rather than on an individual element. In order to better understand the concept of aggregate operations, consider the following example in which we are displaying the capital of countries with their country code:

```
public class CountriesCapital {
  Integer CountryCode;
  String CountryName;
  String CapitalName;
  Integer States;

  CountriesCapital() {}
  CountriesCapital(int cCode, String cName, String capName, int states) {
    this.CountryCode = cCode;
    this.CountryName = cName;
    this.CapitalName = capName;
    this.States = states;
  }
  public String toString() {
    return "" + CountryCode + ", " + CountryName + ", " + CapitalName + ", " + States;
  }
}
```

Let's create a class with the name CountryCapitalDemo which uses the earlier created CountriesCapital class in it:

```
import java.util.*;
import java.util.stream.*;
public class CountryCapitalDemo {
  private static List<CountriesCapital> countriesCapital = Arrays.asList(
      new CountriesCapital(91, "India", "Delhi", 36),
      new CountriesCapital(1, "USA", "Washington", 50),
      new CountriesCapital(43, "Austria", "Vienna", 40),
      new CountriesCapital(86, "China", "Beijing", 55),
      new CountriesCapital(81, "Japan", "Tokyo", 50),
      new CountriesCapital(61, "Australia", "Canberra", 8),
      new CountriesCapital(33, "France", "Paris", 27),
      new CountriesCapital(64, "New Zealand", "Wellington", 53),
```

```
          new CountriesCapital(94, "Sri Lanka", "Colombo", 22)
        );
      public static void main(String[] args) {
          System.out.println("Output with Lambda Expressions:");
          withLExpression();
      }
      private static void withLExpression() {
        List<CountriesCapital> cc = countriesCapital.stream().filter(c -> c.CountryCode >
40).collect(Collectors.toList());
        printList(cc);
      }
      private static void printList(List<CountriesCapital> cc) {
        cc.forEach(c -> System.out.println("\t" + c));
        System.out.println();
      }
    }
```

Here's the output of this example:

```
    Output with Lambda Expressions:
      91, India, Delhi, 36
      43, Austria, Vienna, 40
      86, China, Beijing, 55
      81, Japan, Tokyo, 50
      61, Australia, Canberra, 8
      64, New Zealand, Wellington, 53
      94, Sri Lanka, Colombo, 22
```

Using the **java.util.function** Package

It is aforementioned that the java.util.function package includes interfaces that can be used by the JDK and with the code developed by a user. This package also includes enough functionality that can fulfill the common requirements of the lambda expressions. Consider the following example in which the `java.util.function` package is used:

```
    import java.util.List;
    import java.util.ArrayList;
    import java.time.chrono.IsoChronology;
    import java.time.LocalDate;

    public class Employee {

        public enum Sex {
            M, F
        }

        String empName;
        LocalDate empBirthday;
        Sex empGender;
        String empEmailAddress;

        Employee(String name, LocalDate bday, Sex gen, String email) {
            empName = name;
            empBirthday = bday;
            empGender = gen;
            empEmailAddress = email;
        }

        public int getAge() {
            return empBirthday
                .until(IsoChronology.INSTANCE.dateNow())
                .getYears();
```

```
        }

        public void printPerson() {
          System.out.println(empName + ", " + this.getAge());
        }

        public Sex getGender() {
            return empGender;
        }

        public String getName() {
            return empName;
        }

        public LocalDate getBirthday() {
            return empBirthday;
        }

        public static int compareByAge(Employee a, Employee b) {
            return a.empBirthday.compareTo(b.empBirthday);
        }

        public static List<Employee> createRoster() {

            List<Employee> roster = new ArrayList<>();
            roster.add(
                new Employee(
                "Deepak",
                IsoChronology.INSTANCE.date(1987, 11, 20),
                Employee.Sex.M,
                "deepak.sharma@dreamtechpress.com"));
            roster.add(
                new Employee(
                "Puneet",
                IsoChronology.INSTANCE.date(1986, 7, 15),
                Employee.Sex.M, "puneet@dreamtechpress.com"));
            roster.add(
                new Employee(
                "Nikhil",
                IsoChronology.INSTANCE.date(1991, 8, 13),
                Employee.Sex.M, "nikhil@dreamtechpress.com"));
            roster.add(
                new Employee(
                "Hari Narayan",
                IsoChronology.INSTANCE.date(1991, 8, 18),
                Employee.Sex.M, "hari.narayan@dreamtechpress.com"));

            return roster;
        }

    }
```

Let's now create a class with the name ReferenceMethodsDemo which uses the `Employee` class created earlier:

```
    import java.util.*;
    import java.util.function.Supplier;

    public class ReferenceMethodsDemo {
        public static <T, SOURCE extends Collection<T>, DEST extends Collection<T>>
            DEST transElmnts(SOURCE sCollection, Supplier<DEST> cFactory) {

                DEST res = cFactory.get();
```

```
                for (T tt : sCollection) {
                    res.add(tt);
                }
                return res;
        }

        public static void main(String... args) {

            List<Employee> roster = Employee.createRoster();

            for (Employee e : roster) {
                e.printPerson();
            }

            Employee[] rstrArray =
                roster.toArray(new Employee[roster.size()]);

            class EmployeeAgeComparator
                implements Comparator<Employee> {
                public int compare(Employee e, Employee ee) {
                    return e.getBirthday().compareTo(ee.getBirthday());
                }
            }

            Arrays.sort(rstrArray, new EmployeeAgeComparator());

            Arrays.sort(rstrArray,
                (Employee e, Employee ee) -> {
                    return e.getBirthday().compareTo(ee.getBirthday());
                }
            );

            Arrays.sort(rstrArray, Employee::compareByAge);
            class CmprsnProvider {
                public int cmprByEmpName(Employee e, Employee ee) {
                    return e.getName().compareTo(ee.getName());
                }

                public int compareByAge(Employee e, Employee ee) {
                    return e.getBirthday().compareTo(ee.getBirthday());
                }
            }
            CmprsnProvider myComparisonProvider = new CmprsnProvider();
            Arrays.sort(rstrArray, myComparisonProvider::cmprByEmpName);

            String[] stringArray = { "Deepak", "Puneet", "Nikhil", "Amit", "Hari", "Chirag" };
            Arrays.sort(stringArray, String::compareToIgnoreCase);

            Set<Employee> rosterSet = transElmnts(roster, HashSet::new);
            System.out.println("New way of Display:");
            rosterSet.stream().forEach(q -> q.printPerson());
        }
    }
```

Here's the output of the preceding example:

```
Deepak, 27
Puneet, 28
Nikhil, 23
Hari Narayan, 23
New way of Display:
```

```
Deepak, 27
Hari Narayan, 23
Puneet, 28
Nikhil, 23
```

Summary

In this chapter, you have read about the collections framework which included the `Collection` interface and collection classes. You have also come across generics in Java.

In the next chapter, you'll read about creating packages and interfaces. You'll also come across jar files and annotations used in Java.



22

Creating Packages, Interfaces, JAR Files, and Annotations

If you need an immediate solution to:	*See page:*
Creating a Package	832
Creating Packages that have Subpackages	832
Creating an Interface	833
Implementing an Interface	834
Extending an Interface	835
Using Interfaces for Callbacks	835
Performing Operations on a JAR File	836
Marker Annotations	842
Single Member Annotations	843

In Depth

In this chapter, we are going to take a look at several important Java topics—creating packages, interfaces, JAR files, and annotations. All these topics are popular and good to have under your belt.

Packages and Interfaces

We have been using packages and interfaces throughout the book, and in this chapter, we'll look at how to create them. Being able to create your own packages is a useful skill—one that allows you to divide class libraries into smaller units, much as Oracle has done with the Java packages. You'll see how to create packages in code and how to arrange class files on the disk to match package structure. We'll also take a look at how to create interfaces, which is Java's answer to multiple inheritance.

JAR Files

To make it easier to download class files from the Web, Java supports Java Archive (JAR) files. The archives pack files using the ZIP format, and you'll see how to create JAR files and use them with the JAR tool that comes with Java. In addition, JAR files are the foundation of other parts of Java, such as Java beans.

The Core Java API Package

All of the core API packages defined by Java and their functioning are given in Table 22.1:

Table 22.1: The core Java API packages

Package	Primary function
`java.applet`	It provides supports for applets constructions
`java.awt`	It provides Graphical User Interfaces capabilities
`java.awt.color`	It provides supports for color spaces and profiles
`java.awt.datatransfer`	Data transfers to and from the system clipboard
`java.awt.dnd`	It provides supports for drag-and-drop operations
`java.awt.event`	It handles events
`java.awt.font`	It represents various types of fonts
`java.awt.geom`	It provides the facility to work with geometric shapes
`java.awt.im`	It allows input of Chinese, Japanese, and Korean characters to text editing components
`java.awt.im.spi`	It supports alternative input devices
`java.awt.image`	It processes images
`java.awt.image.renderable`	It supports rendering- independent images
`java.awt.print`	It provides general print capabilities
`java.beans`	It allows you to build software components
`java.beans.beancontext`	It provides an execution environment for Beans
`java.io`	It inputs and outputs data
`java.lang`	It provides core functionality
`java.lang.annotation`	It provides supports for annotations
`java.lang.instrument`	It provides supports for program instrumentation
`java.lang.invoke`	It provides dynamic language support provided directly by the Java core class libraries and virtual machine

Table 22.1: The core Java API packages

Package	Primary function
`java.lang.management`	It provides supports for management of the execution environment
`java.lang.ref`	It enables some interaction with the garbage collector
`java.lang.reflect`	It analyzes code at runtime
`java.math`	It handles large integers and decimal numbers
`java.net`	It supports networking
`java.nio`	Top-level package for the NIO classes and encapsulates buffers
`java.nio.channels`	It encapsulates channels, used by the NIO system
`java.nio.channels.spi`	It supports service providers for channels
`java.nio.charset`	It encapsulates character sets
`java.nio.charset.spi`	It supports service providers for character sets
`java.nio.file`	It defines interfaces and classes for the JVM to access files, file attributes, and file systems.
`java.nio.file.attribute`	Interfaces and classes providing access to file and file system attributes
`java.nio.file.spi`	It provides service-provider classes for the java.nio.file package
`java.rmi`	It provides remote method invocation
`java.rmi.activation`	It activates persistent objects
`java.rmi.dgc`	It manages distributed garbage collection
`java.rmi.registry`	It maps names to remote object references
`java.rmi.server`	It supports remote method invocation
`java.security`	It handles certificates, keys, digests, signatures, and other security functions
`java.security.acl`	It manages access control lists
`java.security.cert`	It parses and manages certificates
`java.security.interfaces`	It defines interfaces for Digital Signature Algorithm (DSA) keys
`java.security.spec`	It specifies keys and algorithm parameters
`java.sql`	It communicates with a Structured Query Language (SQL) database
`java.text`	It provides the facilities for formatting, searching, and manipulating texts
`java.util`	It contains common utilities
`java.util.concurrent`	It supports the concurrent utilities
`java.util.jar`	It creates and reads JAR files
`java.util.logging`	It supports logging of information relating to user preference
`java.util.prefs`	It encapsulates information relating to user and system preferences
`java.util.regex`	It supports regular expression processing
`java.util.spi`	It supports service providers for the utility classes in `java.util` (Added by Java SE 6)

Table 22.1: The core Java API packages	
Package	**Primary function**
`java.util.stream`	It provides classes to support functional-style operations on streams of elements, such as map-reduce transformations on collections
`java.util.zip`	It reads and writes compressed and uncompressed ZIP files

The java.lang Package

The `java.lang` package offers classes that are primarily required for the designing of Java programming language. Java defines the following subpackages:

- java.lang.annotation
- java.lang.instrument
- java.lang.management
- java.lang.reflect
- java.lang.invoke
- java.lang.ref

The *java.lang.annotation* Subpackage

Java's new annotation is supported by the `java.lang.annotation` subpackage. It defines the `Annotation` interface and the `ElementType` and `RetentionPolicy` enumerations.

The *java.lang.instrument* Subpackage

The `java.lang.instrument` subpackage is used for defining the characteristics that you can use for adding instrumentation to several aspects of execution of a program. It defines the `Instrumentation` and `ClassFileTransformer` interfaces, and the `ClassDefinition` class.

The *java.lang.management* Subpackage

The `java.lang.management` subpackage offers execution environment and management supports for the JVM. You can verify and manage various aspects of program execution by using the features available in the `java.lang.management package`.

The *java.lang.reflect* Subpackage

The `java.lang.reflect` subpackage offers interfaces and classes for obtaining reflective information about them. Reflection is the ability of software to examine itself. It is an important feature, especially when using components called Java Beans. It allows you to monitor a software component and describe its capabilities dynamically at runtime rather than at compile time. There are 10 classes in this package. They are listed in Table 22.2:

Table 22.2: Classes of the java.lang.reflect subpackage	
Class	**Does this**
`AccesssibleObject`	It is the base class for Field, Method and `Constructor` objects
`Array`	It provides static methods to dynamically create and access Java arrays
`Constructor<T>`	It provides information about a constructor
`Executable`	It is used as a superclass for the common functionality of Method and Constructor.
`Field`	It provides information about, and dynamic access to, a single field of a class or an interface
`Method`	It provides information about a method

Table 22.2: Classes of the java.lang.reflect subpackage	
Class	**Does this**
`Modifier`	It provides details about class and member access modifiers
`Parameter`	It provides information about method parameters
`Proxy`	It supports dynamic proxy classes
`ReflectPermission`	It allows reflection of private or protected members of class

The *java.lang.invoke* Subpackage

The `java.lang.invoke` subpackage includes support for dynamic language offered directly by the Java core class libraries and virtual machine.

As we have explained earlier in the Java Virtual Machine Specification, there are several types included in this package that have special relations to dynamic language support in the virtual machine:

- The class `MethodHandle` includes signature polymorphic methods. You can connect these methods irrespective of their descriptor type. Mainly, the linkage of method requires exact matching of type descriptors.
- Immediate constants of the classes `MethodHandle` and `MethodType` are supported by the JVM bytecode format.

The *java.lang.ref* Subpackage

The `java.lang.ref` subpackage supports reference-object classes that have interaction with the garbage collector up to a certain limit. The programs may use a reference object to establish references with various other objects in a manner that the latter object may still be retrieved by the collector. The programs may also organize to be notified later when the collector has analyzed that the reachability of a given object has altered.

Basics of Annotation

The Java platform provides an annotation (also known as Metadata) facility that permits defining and use of your own annotation types. It has specific syntaxes for declaring annotation types, for annotating declarations, APIs for reading annotations, a class file representation for annotations, and an annotation-processing tool.

Although annotations do not affect the program semantics directly, they do affect the way used by tools and libraries for treating programs, and that may affect the semantics of the running program. Annotations may be reflected at runtime or may be read from the source files and/or class files. These also complement the use of `javadoc` tags. A `javadoc` tag or an annotation is generally used for markup intended to produce or affect documentation.

Annotation type declarations and normal interface declarations are quite similar. As in the interface keyword, an at sign (@) precedes the annotation type declarations, which is further followed by a parenthesized list of element-value pairs. Each method defined by the declaration defines an element of the annotation type. They must not have any parameters or throws clause. Return types for this declaration are limited to `String`, `primitives`, `enums`, `Class`, `annotations`, and `arrays` of the preceding types. Methods can have default values. Shown here is an example of annotation type declaration:

```
public @interface RequestForEnhancement
{
  int identification();
  String abstract();
  String operator() default "[unassigned]";
  String date() default "[unimplemented]";
}
```

After defining an annotation type, the annotate declarations can be made. You can use annotation anywhere like other modifiers, such as `public`, `static`, or `final`, as it is a special kind of modifier. Conventionally, other modifiers are preceded by annotations. These values must be compile-time constants.

Even though annotations cannot include the `extends` clause, they automatically extend the `Annotation` interface. Thus, `Annotation` can be termed as the super-interface and is declared in the `java.lang.annotation` package. Just look at this code:

```
@classic(txt = "Know me", uni = 224)
public static void Classical() { . . . }
```

In the example, you'll notice that within an annotation (`@classic`), its members are given values. This annotation is linked with the method `Classical()`.

Other Built-In Annotations

Java defines 11 built-in annotations. From them, six were imported from `java.lang.annotation` and five more were included from `java.lang`:

- @Retention
- @Documented
- @Target
- @Native
- @Repeatable
- @Retention
- @Inherited
- @Overdrive
- @Deprecated
- @SuppressWarnings
- @FunctionalInterface
- @ SafeVarargs

@Retention Annotation

The `@Retention` annotation can only be used as an annotation to another annotation. Moreover, it provides the retention policy for Java. A retention policy can determine the point with the help of which an annotation can be discarded. The `java.lang.annotation.RetentionPolicy` class includes three such policies:

- **Retention policy of** `SOURCE` — An annotation of this policy is not included during compilation and retained only in the source file.
- **Retention policy of** `CLASS` — During compilation, an annotation of this policy gets stored in the class file and does not have availability during runtime through the JVM.
- **Retention policy of** `RUNTIME` — In the class file, an annotation of this policy gets stored and is also available at runtime through the JVM.

```
@Retentation (RetentionPolicy.RUNTIME)
@interface classica { }
```

In short, it can be said that the new Metadata facility lets you embed annotation into a program. Various programming tools, in turn, process the annotations. For example, a tool might generate Java source code as requested by an annotation. In the Metadata facility, a programmer describes an action, but relies on a tool in actually providing the code. This type of approach reduces the amount of repetitious code that a programmer may be required to enter manually resulting in considerable time saving.

@Documented Annotation

The `@Documented` annotation can be defined as a marker interface that informs a tool about the documentation of an annotation. It can only be used as an annotation to an annotation declaration.

@Target Annotation

The `@Target` annotation defines the types of declaration to which an annotation can be applied. It is used as annotation for other annotations as well. The `@Target` annotation takes on constant argument from the

ElementType enumeration. This argument specifies the types of declaration to which the annotation can be applied. Table 22.3 shows the constants along with the type of declaration to which they correspond:

Table 22.3: The Target constants and their type of declaration

Target constant	Target annotation
ANNOTATION_TYPE	Another annotation
CONSTRUCTOR	Constructor
TYPE	Class, interface, or enumeration
FIELD	Field
LOCAL_VARIABLE	Local variable
PARAMETER	Parameter
PACKAGE	Package
METHOD	Method

In a @Target annotation, you can mention one or more of these values. You must specify multiple values inside a braces-delimited list. For example, to specify that a @Target annotation applies only to fields and local variables:

```
@Target ({ ElementType.FIELD, ElementType.LOCAL_VARIABLE })
```

@Native Annotation

The @Native annotation specifies a field that defines a constant value which can be referenced from native code. This annotation can be used as a hint by tools that are used for generating native header files. These files are used for determining whether a header file is needed or not, and if needed, then what declarations it should have.

@Repeatable Annotation

The @Repeatable annotation is used for indicating that an annotation type whose declaration it (meta-) annotates is repeatable. The value of the @Repeatable annotation specifies the stored annotation type for the repeatable annotation type.

@Retention Annotation

The @Retention annotation specifies the time of retention of annotations having annotated type are to be retained. The default value of the retention policy is RetentionPolicy.CLASS in case no Retention annotation is available on as annotation type declaration.

The @Retention meta-annotation is affected only if you use the meta-annotated type directly for annotation.

@Inherited Annotation

The @Inherited annotation can be referred to as a marker annotation that can only be used on declaration of another annotation. In addition to this, it only affects that annotation which is used on class declaration. The @Inherited annotation allows the superclass annotation to be inherited by a subclass. Therefore, whenever a request is made for a particular annotation to the subclass, the superclass is checked if the annotation for the subclass is not available. Suppose that annotation is available in the superclass which is annotated with the @inherited keyword, then an annotation is returned.

@Override Annotation

The @Override annotation is used only on methods and can be referred to as a marker annotation. A method annotated with the @ Overdrive keyword needs to override a method from a superclass. If it does not override a method, a compile-time error appears. This annotation can be used for ensuring overriding of a superclass method.

@Deprecated Annotation

The `@Deprecated` annotation specifies that a declaration is removed and replaced by a newer form.

@SuppressWarnings Annotation

The `@SuppressWarnings` the annotation specifies suppressing of one or more notifications that might be provided by the compiler. The suppressed notifications are required to be mentioned by names in the form of string. You can apply this annotation to any type of declaration.

@FunctionalInterface

An informative annotation type is used for specifying an interface type declaration that is intended to be a functional interface. Generally, a functional interface contains has exactlyonly one abstract method. Since Default methods cannot be treated as abstract as they have an implementation.

You can create instances of functional interfaces by using lambda expressions, method references, or constructor references.

Compilers must generat an error message, if a type is annotated with this annotation type, unless the following conditions are satisfied:

- The type is an interface type and not an annotation type, enum, or class.
- The annotated type fulfills the need of a functional interface.

The compiler treats any interface satisfying the definition of a functional interface as a functional interface sirrespective of whether or not a `@FunctionalInterface` annotation is available on the interface declaration.

@ SafeVarargs

Application of the @SafeVarargs annotation to a method or constructor suppresses unchecked notifications about a non-reifiable variable arity (vararg) type and creation of parameterized array creation at call sites.

In addition to the imposing of usage restrictionsconstraints imposed applied by its HYPERLINK "http://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Target.html" \o "annotation in java.lang.annotation" @Target meta-annotation, compilers are need to implement more usage constraints on this annotation type. It is a compile-time error in case of declaration of a method or a constructor annotated with a @SafeVarargs annotation, and either:

- the declaration is a fixed arity method or constructor, or
- the declaration is a variable arity method neither static nor final.

Compilers issue notifications when this annotation type is used for to a method or constructor declaration, where the following conditions hold:

- The element type of the variable arity parameter is reifiable which includes primitive types: Object, and String.
- Unsafe operations are performed by the body of the method or constructor. Array Sometimes, unchecked warnings are not generated by some unsafe operations. For example, consider the following code snippet:

```
@SafeVarargs // Not actually safe!
static void m(List<String>... stringLists) {
  Object[] array = stringLists;
  List<Integer> tmpLlist = Arrays.asList(42);
  array[0] = tmpLlist; // Semantically invalid, but compiles without warnings
  String str = stringLists[0].get(0); // Oh no, ClassCastException at runtime!
}
```

The aliasing in the preceding code snippet leads to the `ClassCastException` exception at runtime. The statement arr[0]=tmplst is semantically invalid, but will execute without warnings. However, the statement String str=stringLists[0].get(0) gets executed and will give the `ClassCastException` exception.

Type Annotations and Pluggable Type Systems

Before the release of Java SE 8, you could only apply annotations to declarations. But the release of Java SE 8has let you apply annotations to any type. Some instances where types are used are `implements` clauses, object creation expressions (`new`), and `casts` and `throws` clauses. This kind of annotation is known as type annotation.

Type annotations allow better analysis of Java programs and ensure stronger type checking. A type checking framework is not provided in Java SE 8, but it enables you to write (or download) a type checking framework. This framework can be implemented as single or multiple pluggable modules that can be used in conjunction with the Java compiler.

Repeating Annotations

In Java SE8, you can use repeating annotations in situations where you want to apply the same annotation to a declaration or type use. Suppose you want to use a timer service which allows you to execute a method at a specified time like the UNIX `cron` service. For creating this timer service, you need to write a code. Using the code, you can set a timer for executing a method, `doPeriodicCleanup`, on every month's last day and on every Saturday at 10:00 p.m. Now, in order to set the timer for execution, you need to write an `@Schedule` annotation and use it twice for the `doPeriodicCleanup` method: first for specifying the last day of the month and the second for specifying Saturday at 10 p.m., as shown in the following statements:

```
@Schedule(dayOfMonth="last")
@Schedule(dayOfWeek="Sat", hour="23")
public void doPeriodicCleanup() { ... }
```

In the preceding statements, an annotation to a method is applied. Now, you can repeat an annotation anywhere so that you will be able to use a standard annotation. The repeating annotations are stored in a *container annotation* that is generated automatically by the Java compiler for compatibility reasons. For making the compiler to do this, the following two declarations are needed in your code:

- The annotation type must be specified with the `@Repeatable` meta-annotation
- The containing annotation type must contain a value element with an array type

Retrieving Annotations

Reflection API provides various methods that can be used for retrieving annotations. The behavior of these methods remains unchanged, which return a single annotation, such as AnnotatedElement.getAnnotationByType`(Class<T>)`, if there is availability of one annotation of the requested type. In case there is availability of more than one annotation of the requested type, you can retrieve them by getting their container annotation first. In this way, legacy code functions continuously. Other methods were also introduced in Java SE 8 for scanning through the container annotation for returning multiple annotations together, such as AnnotatedElement.getAnnotations`(Class<T>)`.

That's it for the overview of what is in this chapter. There's a great deal coming up, so it's time to turn to the "Immediate Solutions" section.

Immediate Solutions

Creating a Package

Good Lord! says the Novice Programmer, "I have so many classes that things are getting pretty cluttered now. How can I wade through all this mess?" "Well," you say, "why not divide your classes into packages? That will let you arrange your class files in a directory structure much like you'd arrange files when you have too many of them. Problem solved." "Great," says the NP, "tell me how it works."

When you have a number of class files, it's often a good idea to arrange them on disk by using a directory hierarchy. In fact, Java packages were originally designed to reflect such a file organization. You can arrange class files into a directory hierarchy and let Java know what's going on with the packages. For example, if you have a package named `package1` with one class in it, `app.class`, the class file would go into a directory named `package1`, like this:

```
package1
   |____app
```

Multiple class files in the same package go into the same directory:

```
package1
   |____app1
         |____app2
               |____app3
```

As long as the `package1` directory is in a location that Java will search (see Chapter 1 for more information on where Java will search for class files and how to use the `CLASSPATH` environment variable), Java will look for the class files you list as part of that package in that directory.

You need to indicate what package a class file is part of by using the `package` statement in your code. Here's an example where we are creating `app.class` and indicating that it's part of `package1` by using the `package` statement:

```
package package1;
public class app {
         public static void main(String[] args) {
   System.out.println("Hello from Java!"); }
}
```

After we have compiled `app.class` and stored it in the `package1` directory, we can import the `app` class into my code with statements such as the following, just as we would with any other Java package:

```
import package1.*;
or
import package1.app;
```

The Java runtime tool also knows about packages; since `app.class` is itself an application, we can run this class with a command line that specifies the `app` class's package by using a dot (`.`) for the directory separator:

```
E:\Java8\chapter22>java package1.app
Hello from Java!
```

In fact, the dot package separator comes in handy when you create subpackages of packages—take a look at the next solution for the details.

Creating Packages that have Subpackages

"Hmm," says the Novice Programmer, "I understand that you can put class files into a package by setting up the correct directory structure and by using the `package` statement, but what if the packages have subpackages? How deep can I nest the package structure?" "As deep as you want," you say, "that is, as deep as your operating system supports nested directories."

When you have a lot of class files, you may want to organize them into quite a complex package structure, and you do that by creating the corresponding directory structure on disk, including giving and subdirectories. For example, if you want the `app` class to be in `package2`, which itself is a subpackage of `package1`, this is the directory structure you'd use:

```
package1
   |____package2
         |____app
```

To create this package structure in code, you just use the dot package separator. Here's an example:

```
package package1.package2;

public class app
{
   public static void main(String[] args)
   {
   System.out.println("Hello from Java!");
   }
}
```

Now you can import the `app` class into your code with statements such as these:

```
import package1.package2.*;
or
import package1.package2.app;
```

Since here `app` is already an application class, you can also run it like this, specifying the package structure:

```
E:\Java8\chapter22>java package1.package2.app
Hello from Java!
```

Creating an Interface

"Well," the NP says, "I want to derive a new class from two base classes, but I know that Java doesn't support multiple inheritance, so...." "So you have to use interfaces," you say.

We have used interfaces throughout the book, and we'll take a look at how to create one here. When you create an interface, you specify the methods for the interface, and when you implement the interface, you provide code for those methods. Interfaces have two advantages over inheritance:

- A class can extend to only one other class, but as many interfaces as you need.
- They are easier to work with than interfaces because you don't have to provide any implementation details in the interface.

NOTE

When an interface is declared as public, it can be used by other classes as well.

Here's an example in which we create an interface named `Printem` that has one method: `printText()`. We implement this interface in a class named `class1`. To create an interface, we simply use the `interface` statement and list the prototypes (declarations without bodies) of the methods we want in the interface, like this:

```
interface Printem { void printText(); }
```

Now we can implement this interface in `class1`:

```
interface Printem { void printText(); }

class interfaces
{
   public static void main(String[] args)
   {
         class1 object1 = new class1();
         object1.printText();
   }

}
class class1 implements Printem
{
   public void printText() { System.out.println("Dreamtech Press"); }
}
```

The output of the code is shown in Figure 22.1:

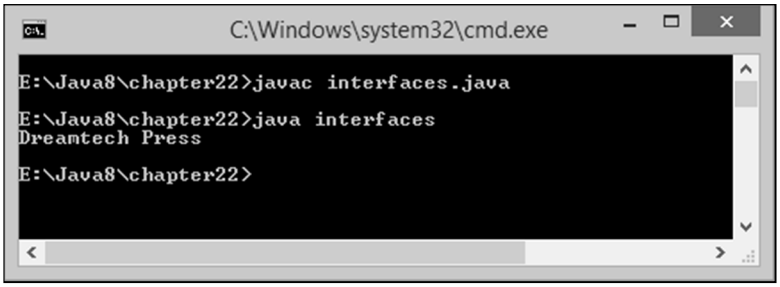


Figure 22.1: Printing text using interface

Implementing an Interface

You can create classes that partly implement an interface, but these classes must be declared abstract because they are only partial implementations and therefore cannot be instantiated into objects. Here's an example in which we create an interface with two methods: `printText1()` and `printText2()`:

```
interface Printem { void printText1(); void printText2(); }
```

Next, we create a new class, `Class1`, that implements the interface but only defines one method, `printText1()` (note that we must declare this class abstract as it is not a full implementation of the interface):

```
interface Printem { void printText1(); void printText2(); }
abstract class Class1 implements Printem
{
  public void printText1() { System.out.println("Hello from Java!"); }
}
```

Finally, we can create a new class, `Class2`, that extends `Class1` and defines the `printText2()` method, which means we can instantiate objects of `Class2` as shown here:

```
interface Printem { void printText1(); void printText2(); }

abstract class Class1 implements Printem
{
  public void printText1() { System.out.println("Hello from Dreamtech Press"); }
}

class Class2 extends Class1
{
  public void printText2()
  {
        System.out.println("Hello from Java interfaces!");
  }
}

public class Interfaces2
{

  public static void main(String[] args)
  {
        Class2 object2 = new Class2();
        object2.printText1();
        object2.printText2();
  }

}
```

The output of this code is shown in Figure 22.2:

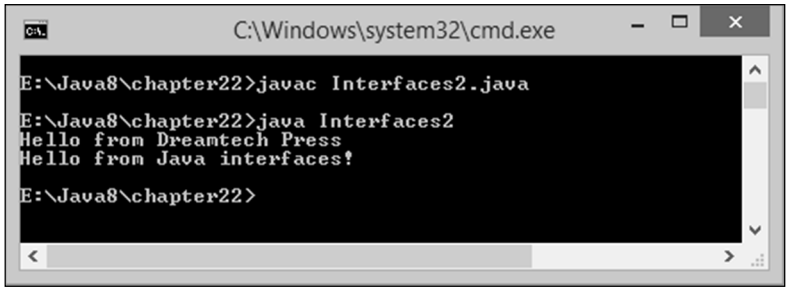


Figure 22.2: Implementing an interface

Note that partial implementations can be useful, such as when you want to specify how some of an interface's methods, but not all, should work in a series of subclasses.

In fact, an interface is a kind of type, just like class. As a result, you can use an interface as the type for a variable, parameter, or method return value. For example, consider this interface:

```
public interface Play { void play(); }
. . .
Play gme = getGame();
gme.play();
. . .
```

Here the method `getGame()` returns an object that implements the `Play` interface and this object is assigned to a variable of type `Play`. Then the object's `play()` method is called.

Extending an Interface

Interfaces are extended with the use of `extends` keyword. An interface that is getting extended is known as the *superinterface*; on the other hand, the interface which is extending an existing interface is known as the *subinterface*.

```
public interface color
{
  void blue();
  void red();
}
public interface shirt
{
  void sz1();
  void sz2();
}
public interface combination extends color, shirt { void combin1(); }
```

When the `extends` keyword is used with interfaces, all the fields of the superinterface are copied into the subinterface. In other words, three interfaces are declared—the first defines two methods `blue()` and `red();` the second defines two methods `sz1()` and `sz2();` and the third extends `color` and `shirt`, and adds a method of its own called `combin1()`.

Using Interfaces for Callbacks

A callback is a technique wherein an object lets another object know that the second object should call one of the first object's methods whenever a certain event happens. The first object used in this technique is known as an event listener as it waits till it is called by the other object. The second object in this technique is known as the event source as it is the source of events that provides calls to the listener.

Such callbacks are handled in Java through Interfaces. The callbacks are generally used in graphical Swing applications, where event listeners are built for handling the events of user interface, such as mouse clicks.

Let's look at this example where we have implemented callbacks. In this example, a timer is implemented in the class Ticking. This timer generates events after regular intervals and sets up a listener object for handling these events:

```
import java.awt.event.*;
import javax.swing.*;

public class Ticking
{

  public static void main(String[] args)
  {
        Timer tm = new Timer(1000, new Ticker());
        tm.start();
        JOptionPane.showMessageDialog(null, "You wish to exit...cool...Click OK"); }
}

class Ticker implements ActionListener
{
  private boolean tick = true;

  public void actionPerformed(ActionEvent event)
  {
        if (tick) { System.out.println("Tick..."); }
        else {  System.out.println("Tock..."); }
        tick = !tick;
  }

}
```

Here we have used the `ActionListener` interface that consists of a single method named `actionPerformed()`. A parameter of type `ActionEvent` is received by the `actionPerformed()` method. The result displays the messages Tick… and Tock…. on the console, as shown in Figure 22.3:



Figure 22.3: Displaying Tick and Tock

Performing Operations on a JAR File

A Java Archive (JAR**)** file comprises various directories and isolated files that are saved in a condensed format in zip pattern. The .jar is the extension for JAR files. A JAR file is unzipped using software and then all its contents are extracted.

Also, many processors, such as the Java Virtual Machine, or browsers, can use the files in a JAR directly without extracting the files.

Creating a JAR File

"Wow," says the Novice Programmer, "my class files are getting huge and there are a lot of them. Do you have any idea what that does to the download times for my applets?" You smile and say, "I sure do—which is why you should be using JAR files." "*What* files?" the NP asks.

A JAR file allows you to wrap various files into one. In fact, JAR files compress their contents by using the ZIP format, so if your applet needs a lot of large files, it makes sense to create and download a JAR file. Web browsers only need one connection rather than new connections for each sub file, which can improve download times. The digital signature is used by the files in a JAR file for proving their origin. You create JAR files with the jar tool, and we'll take a look at that tool in this and the next few solutions. This tool comes with Java, and here's how you use it:

```
jar [options] [manifest] destination input-file [additional input files]
```

Here, *options* refer to the options you can use with the jar tool, and they are much like the options you can use with the UNIX tar tool. The optional *manifest* argument consists of the name of a manifest file, which supports digital signing and lists the contents of the JAR file. When you create a Java bean, you use the manifest file to indicate which classes are beans, as we'll do later in this chapter. Here are the possible options you can use with the jar tool:

- `c`—It creates a new or empty archive on the standard output.
- `t`—It lists the table of contents from standard output.
- `x file`—It extracts all files or just the named files. In case a `file` is not present, then all the files are accessed; else, only the specific files are accessed.
- `f`—The argument following this option specifies a JAR file to work with.
- `v`—It generates "verbose" output on `stderr`.
- `m`—It includes manifest information from a specified manifest file.
- `0`—It indicates "store only," without using ZIP compression.
- `M`—It specifies that a manifest file should not be created for the entries.
- `u`—It updates an existing JAR file by adding files or changing the manifest.
- `-C`—It changes directories during execution of the `jar` command. For example, `jar uf jarfile.jar -C classes` would add all files within the classes' directory, but not the classes' directory itself, to `jarfile.jar`.

TIP

Use an argument beginning with the character "@" to specify a file containing additional arguments, one argument per line. The arguments are inserted into the command line at the position of @filename argument.

Here's a typical usage of the jar tool:

```
E:\Java8\chapter22>jar cf jarfile.jar *.class
```

Here, all the class files present in the current directory are stored in the file with the name `jarfile.jar`. A default manifest file gets generated automatically by the jar tool and is always the first entry in the jar file. If you have a manifest file that you want the jar tool to use for a new JAR archive, you can use the `-m` option and specify it like this (we'll create manifest files later in this chapter):

```
E:\Java8\chapter22>jar cmf manifest.mft jarfile.jar *.class
```

You must note that if you mention the options `cfm` in place of `cmf`, then you must first mention the name of the JAR archive, which will be further followed by the manifest file's name:

```
E:\Java8\chapter22>jar cfm jarfile.jar manifest.mft *.class
```

Note also that JAR files are not just for class files; they can store any type of files. Here's how to pack all the files in a directory into a JAR file:

```
E:\Java8\chapter22>jar cfm jarfile.jar manifest.mft *.*
```

If the file names you want to pack include any directories, those directories are searched recursively and the files in them are stored. When the JAR file is unpacked, the directory structure will be re-created.

Getting the Contents of a JAR File

"Uh-oh," says the Novice Programmer, "I have a JAR file and I have forgotten what's in it. I guess I'll have to rebuild it." "Not at all," you say. "Just use the `tf` options." "How's that?" the NP asks.

You can determine what's in a JAR file with the `tf` options as in this case, where the JAR file contains a default manifest in the `META-INF` internal directory and a number of class files:

```
E:\Java8\chapter22>jar tf SampleJar.jar
```

The output of the command given above will be as follows:

```
META-INF/
META-INF/MANIFEST.MF
applet.class
jpanel.class
newButton.class
textPanel.class
```

Extracting Files from a JAR File

"OK," says the Novice Programmer, "I have been using JAR files to archive my files— but how do I get those files out of a JAR file?" You say, "With the `xf` options." "Great!" says the NP.

You can extract files from a JAR file by using the `xf` options. For example, here's how to extract all the files in the file `jarfile.jar`, introduced in the previous solution:

```
E:\Java8\chapter22>jar xf jarfile.jar
```

Unpacking all the files in a JAR file this way will also re-create their directory structures. In this case, the jar tool unpacks the class files, and also makes a directory named `META-INF` and places the default manifest file `MANIFEST.MF` in it. You can also extract files by specifying their names. Here's an example in which we are extracting `applet.class`:

```
E:\Java8\chapter22>jar xf jarfile.jar applet.class
```

Updating JAR Files

Here's how to update a JAR file with a new version of one of the files stored in it, class1.class, using the `-uf` options:

```
E:\Java8\chapter22>jar -uf jarfile.jar class1.class
```

Reading from JAR Files in Code

"Say," the Novice Programmer says, "now that I have created JAR files, how can I unpack them at runtime from my code? Is there any way?" "Yes," you say, "you can use the `JarFile` class." "Tell me more!" says the NP.

The `JarFile` class can be used for reading the data stored in a JAR file from any file that can be opened using the `java.io.RandomAccessFile` package. The `java.util.zip.ZipFile` class is extended by the `JarFile` class with support to read an optional manifest entry. Here's the inheritance diagram for the `ZipFile` class:

```
java.lang.Object
    |____java.util.zip.ZipFile
```

You'll find the constructors for the `ZipFile` class in Table 22.4 and its methods in Table 22.5:

Table 22.4: Constructors of the ZipFile class

Constructor	Does this
`ZipFile(File file)`	It opens a ZIP file for reading from the indicated File object
`ZipFile(File file, int mode)`	It opens a ZIP file for reading from the specified `File` object in the specified mode
`ZipFile(String name)`	It opens a ZIP file for reading

Table 22.5: Methods of the ZipFile class	
Method	**Does this**
`void close()`	It closes the ZIP file
`Enumeration entries()`	It gets an enumeration of the ZIP file entries
`protected void finalize()`	It ensures that the `close()` method of this ZIP file is called when there are no more references to it
`ZipEntry getEntry (String name)`	It gets the ZIP file entry for the indicated name (or null if not found)
`InputStream getInputStream (ZipEntry entry)`	It gets an input stream for reading the contents of the indicated ZIP file entry
`String getName()`	It gets the pathname of the ZIP file
`int size()`	It gets the number of entries in the ZIP file

The `JarFile` class is built on the `ZipFile` class. Here's the inheritance diagram for the `JarFile` class:

```
java.lang.Object
    |____java.util.zip.ZipFile
        |____java.util.jar.JarFile
```

You'll find the field of the `JarFile` class in Table 22.6, its constructors in Table 22.7, and its methods in Table 22.8:

Table 22.6: The field of the JarFile class	
Field	**Does this**
`static String MANIFEST_NAME`	It holds the name of the manifest

Table 22.7: Constructors of the JarFile class	
Constructor	**Does this**
`JarFile(File file)`	Constructs new JAR file to read from indicated File object
`JarFile(File file, boolean verify)`	It constructs a new JAR file to read from the indicated File object and allows for verification
`JarFile(File file, boolean verify, int mode)`	It constructs a new JAR file to read from the indicated File object in the specified mode
`JarFile(String name)`	Constructs a new JAR file to read from indicated file name
`JarFile(String name, boolean verify)`	It constructs a new JAR file to read from the indicated file name and allows for verification

Table 22.8: Methods of the JarFile class	
Method	**Does this**
`Enumeration<JarEntry> entries()`	It gets an enumeration of the ZIP `file` entries
`ZipEntry getEntry (String name)`	It gets the ZIP `file` entry for the indicated name (or null if not found)
`InputStream getInputStream (ZipEntry ze)`	It gets an input stream for reading the contents of the indicated ZIP `file` entry
`JarEntry getJarEntry (String name)`	It gets the `JarEntry` object for the given entry name (or null if not found)
`Manifest getManifest()`	It gets the JAR file manifest (or null if none)

You actually handle the items in a JAR file as `JarEntry` objects, and the `JarEntry` class is derived from the `ZipEntry` class. Here's the inheritance diagram for `ZipEntry`:

```
java.lang.Object
  |____java.util.zip.ZipEntry
```

You'll find the fields of the `ZipEntry` class in Table 22.9, its constructors in Table 22.10, and its methods in Table 22.11:

Table 22.9: Fields of the ZipEntry class

Field	Does this
`static int DEFLATED`	Specifies the compression method for compressed entries
`static int STORED`	Specifies compression method for uncompressed entries

Table 22.10: Constructors of the ZipEntry class

Constructor	Does this
`ZipEntry(String name)`	It constructs a new ZIP entry with the indicated name
`ZipEntry(ZipEntry e)`	It constructs a new ZIP entry with fields taken from the indicated ZIP entry

Table 22.11: Methods of the ZipEntry class

Method	Does this
`Object clone()`	It gets a copy of this entry
`String getComment()`	It gets the comment string for the entry (or null if none)
`long getCompressedSize()`	It gets the size of the compressed entry data (or -1 if not known)
`long getCrc()`	It gets the CRC-32 checksum of the uncompressed entry data (or -1 if not known)
`byte[] getExtra()`	It gets the extra field data for the entry (or null if none)
`int getMethod()`	It gets the compression method of the entry (or -1 if not indicated)
`String getName()`	It gets the name of the entry
`long getSize()`	It gets the uncompressed size of the entry data (or -1 if not known)
`long getTime()`	It gets the modification time of the entry (or -1 if not indicated)
`int hashCode()`	It gets the hashcode value for this entry
`boolean isDirectory()`	It returns True if this is a directory entry
`void setComment (String comment)`	It sets the optional comment string for the entry
`void setCompressedSize (long csize)`	It sets the size of the compressed entry data
`void setCrc(long crc)`	If sets the CRC-32 checksum of the uncompressed entry data
`void setExtra(byte[] extra)`	It sets the optional extra field data for the entry
`void setMethod(int method)`	It sets the compression method for the entry
`void setSize(long size)`	It sets the uncompressed size of the entry data
`void setTime(long time)`	It sets the modification time of the entry
`String toString()`	It gets a string representation of the ZIP entry

Here's the inheritance diagram for the `JarEntry` class:

```
java.lang.Object
   |____java.util.zip.ZipEntry
          |____java.util.jar.JarEntry
```

You'll find the constructors for the `JarEntry` class in Table 22.12 and its methods in Table 22.13:

Table 22.12: Constructors of the JarEntry class

Constructor	Does this
`JarEntry(JarEntry je)`	Constructs new `JarEntry` object from given `JarEntry` object
`JarEntry(String name)`	It constructs a new `JarEntry` object for the given JAR file entry name
`JarEntry(ZipEntry ze)`	It constructs a new `JarEntry` object from the given `ZipEntry` object

Table 22.13: Methods of the JarEntry class	
Method	**Does this**
`Attributes getAttributes()`	It gets the manifest attributes for this entry (or null if none)
`Certificate[] getCertificates()`	It gets the certificate objects for this entry (or null if none)
`CodeSigner getCodeSigners()`	It gets the `CodeSigner` objects for this entry (or null if none)

Using *javac* to Get Classes from a JAR File

For convenience, you can use the Java compiler, `javac`, to unpack class files from JAR files as needed. Here's an example in which we have a class named `display` with one method, `displayText()`, which displays a message:

```
public class display
{
  public void displayText() { System.out.println("Hello from Java!"); }
}
```

Here's how we make a JAR file from this class:

```
E:\Java8\chapter22>jar cf display.jar display.class
```

Next, we delete the `display.class` file, so all that's left is the JAR file, `display.jar`. Now we can make use of the `display` class in an application, like this:

```
public class useDisplay
{
  public static void main(String[] args)
  {
        display display1 = new display();
        display1.displayText();
  }
}
```

To compile this application, include `display.jar` in your CLASSPATH so that `javac` will search the JAR file for `display.class` and unpack that class. Because `display.class` is now unpacked, it's available when we run the application that uses it:

```
E:\Java8\chapter22>javac -cp E:/Java8/chapter22/display.jar useDisplay.java
E:\Java8\chapter22>java useDisplay
Hello from Java!
```

Letting Applets Get Classes from JAR Files

The NP is back and says, "Now I have put my applet into a JAR file, as well as all the support files it needs. But there's one problem." "What's that?" you ask. The NP says, "How in the world do I tell the browser about JAR files?"

You can tell Web browsers and the Oracle appletviewer that you are storing files in a JAR file with the `archive` parameter in the `<APPLET>` tag. Here's an example. In this case, we are creating an applet that will be stored in a JAR file named `applett.jar`. Here's the code for this applet, including the `<APPLET>` tag:

```
import java.awt.*;
import java.applet.Applet;
/*<APPLET
  CODE=applett.class
  WIDTH=200
  HEIGHT=200><PARAM NAME=archive VALUE="applett.jar"></APPLET> */
public class applett extends Applet
{
  public void paint(Graphics g) { g.drawString("Dreamtech Press", 30, 80); }
}
```

After compiling this applet to `applett.class`, we create the `applett.jar` file:

```
E:\Java8\chapter22>jar cf applett.jar applett.class
```

Next, we delete `applett.class` and open `applett.java` by using the Oracle appletviewer:

```
E:\Java8\chapter22>appletviewer applett.java
```

The result appears as shown in Figure 22.4:



Figure 22.4: Opening an applet from a JAR file

As you can see, the appletviewer has opened the applet from the JAR file.

Marker Annotations

A marker annotation is a special kind of annotation that is used for marking a declaration, and it does not contain any members. The `AnnotedElement` interface defines the `isAnnotationPresent()` method, which denotes whether the marker annotation is present. Since a marker interface contains no members, simply determining whether it is present or not is sufficient. The code to implement the marker annotations is as follows:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface Mark {}
class Marker
{
  @Mark
  public static void Meth ()
  {
        Marker obj = new Marker();
        try {
                Method mt = obj.getClass () .getMethod ("Meth");
                if (mt.isAnnotationPresent (Mark.class))
                System.out.println ("Here is the marker of that application");
                 System.out.println ("Welcome to DreamTech Press");
        }
        catch (NoSuchMethodException exc) {
                System.out.println ("Method Not Found.");
        }
  }
  public static void main (String args[])
  {
        Meth();
  }
}
```

The output shown in Figure 22.5 confirms that `@Mark` is present:

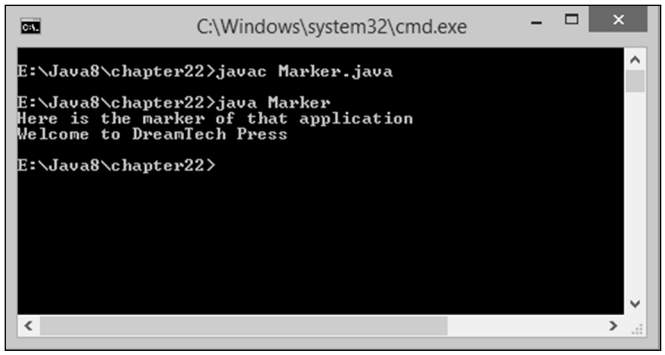


Figure 22.5: Use of marker

Notice that you do not need to follow `@Mark` with parentheses when it is applied. `@Mark` is applied simply by using its name, like this `@Mark`.

NOTE

An empty set of parenthesis can also be supplied.

Single Member Annotations

A single member annotation comprises only one member and functions similar to a normal annotation. It enables a shorthand form for mentioning the member's value. In case only a single member is present, then you do not require to mention the member's name. In fact, you can simply specify the value for that member when the annotation is applied.

See how we use a single-member annotation in the example:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface Sngle { int value(); }
class Single
{
  @Sngle(100)
  public static void Meth()
  {
        Single obj = new Single();
        try { Method mt = obj.getClass().getMethod("Meth");
                Sngle sng = mt.getAnnotation(Sngle.class);
                System.out.println(sng.value()); }
        catch (NoSuchMethodException exc)
        {
                System.out.println ("Method Not Found.");
        }
  }
  public static void main(String args[]) { Meth(); }
}
```

As expected, the result of this program gets displayed as 100. In the program, `@Sngle` is used to annotate `Meth()` as `@Single(100)`. Notice the way a value is inserted. There is no need for 'value = '.

NOTE

In single-member annotation, the name of the member must be value.

Summary

In this chapter, You have learned about the concept of packages, interfaces, and JAR files. Then, you have learned core Java API package and the `java.lang` package. We explored how to create a package, a subpackage; create and implement an interface; and use interfaces for callbacks. Lastly, we have discussed how to perform operations on a JAR file and use annotations.

The next chapter discusses working with Java Beans.



23

Working with Java Beans

If you need an immediate solution to:	*See page:*
Understanding Java Beans	850
Designing Programs Using Java Beans	850
Creating Applets that Use Java Beans	853
Creating a Java Bean	853
Creating a Bean Manifest File	855
Creating a Bean JAR File	855
Creating a New Bean	855
Adding Controls to Beans	855
Giving a Bean Properties	856
Design Patterns for Properties	862
Using Simple Properties	862
Designing Patterns for Events	864
Learning Methods and Design Patterns	865
Creating Bound Properties	865
Giving a Bean Methods	866
Giving a Bean an Icon	867
Creating a BeanInfo Class	867
Setting Bound and Constrained Properties	868
Implementing Persistence	870
Using the Java Beans API	870
Learning the Basics of an Event	872
Using the Java Beans Conventions	873
Using the Remote Notification and Distributed Notification	873
Using Beans with JSP	873

In Depth

Sun provides the JavaBeans Bridge for ActiveX components. Beans are reusable components written in Java that you can use in a variety of programming environments, including third-party programming environments. In fact, they work much like ActiveX controls, exposing properties, methods, and events to other code components. Using beans, you can create your own new Java "controls," such as buttons that turn colors when you click them or stock tickers that download information from the Internet.

Java Beans is designed to be used in application builder tools, which are programming environments that let you configure beans. One such tool, the *BeanBox*, comes with the Sun Bean Development Kit (BDK), which you can download and install from `http://www.ecst.csuchico.edu/~amk/foo/csci611/notes/beans/BDK/`. We'll use the BDK in this chapter to construct and use beans. As reusable code components that you can create applications from, beans represent a powerful new way of looking at programming—you can create your own applications on the fly from pre-written components by adding just what you need—and nothing more.

Java Beans has a very important part for the Java users. This part is very important. The reason for their importance is they allow you to build complex systems from software components. Usually, there are different vendors who supply these components, and sometimes, these components are also made available by you. Java Beans defines the structural design that shows how the different building units work together.

While using a component-based approach, the software components can be easily reused and interoperability between the components can also be made possible. This approach allows the programs to be collected from software building blocks provided by different vendors to use the benefits. It is necessary for a designer to select a component, understand it, and also check for its availability. Then, it becomes easy to implement its operation into the existing code. Such kind of architecture is provided by Java Beans.

Oracle supports Java Beans in the `java.beans` package. We will discuss about the interfaces and classes under this package in the *Immediate Solutions* section.

In this chapter, we'll create a number of Java Beans—from simple ones to ones that support properties and methods and let users embed other controls inside them.

TIP

The way you create beans varies a little among operating systems because the BDK differs a little among operating systems. We use the BDK for Windows in this chapter.

What is Java Bean?

A Java Bean refers to a software component that can be reused in different environments. It can perform different types of function, i.e., simple as well as complex. Simple function examples include obtaining an inventory value, and complex function examples include forecasting the performance of a stock portfolio. A Bean may be visible to an end user. An example of this is using a radio within a car, which is also used at homes; thus, it serves the purpose at various places.

A Bean may also be visible to a user. An example of this type of building block is software that encodes a stream of data in real time. Finally, a Bean can work separately on a user's computer or in association with a group of distributed components. An example of a Bean that runs locally is the software that creates a bar chart from a set of data points. However, it is necessary that a Bean which provides real-time price information from a stock or commodities exchange would need to work in corporation with other distributed software to obtain its data.

The builder tools use a process for discovering the different features, such as methods, events, and properties, of a Bean. This process is called introspection. The following are the features of Beans:

- It provides support by adhering to specific rules, known as design patterns, at the time when naming Bean features. The `Introspector` class monitors Beans for such design patterns to explore Bean features. This class relies on the core reflection API. The Reflection API is an excellent place to learn about reflection.

- It provides support by explicitly providing property, method, and event information with a related `Bean information` class. This Bean information class executes the BeanInfo interface. Also, the Bean information class explicitly records the Bean features that are to be used for application builder tools.
- Properties are the Bean's look and its behavior, which can be modified at design time. Builder tools look into a Bean to discover its properties and expose those properties for calculation.
- The properties of Beans are customized at design time. Here, customization is achieved through two methods, that is, by using property editors and by using more technical Bean customizers.
- Beans use events in order to communicate with other Beans. Beans are of two types, namely, listener Bean and source Bean. Listener Bean is a Bean that shows its interest in receiving events and registers its interest with the source Bean. The source Bean is the Bean that fires the event. A Bean is monitored by Builder tools to ascertain which events are sent and which events can be handled by the bean. *Persistence* enables Beans to save and restore their state. Once we modify the properties of Beans, we can save the Bean's state and resume it later. Remember that property changes remain intact. Java Beans use Java Object Serialization to support persistence. A Bean's method can be invoked from other Beans or a scripting environment. The methods are similar to Java's methods. Beans are created to be understood by builder tools and all key APIs, including properties, events, and persistence, meant to be easily understandable by programmers. The component technology is brought to the Java platform by the Java Beans. You can create reusable, platform-independent components with the Java Beans API. All the components can be combined with the use of Java Beans-compliant application builder tools into applets, applications, or composite components. Beans are known as Java Bean components.

Advantages of Java Bean

The Java Bean technology plays a very important role. Some of the benefits that Java Bean technology provides for a component developer are:

- All the benefits of Java's "write-once, run-anywhere" paradigm are present in the Bean.
- The properties, events, and methods of a Bean are viewed by another application, such as a builder tool, which can be monitored.
- To configure a Bean, virtual software can be used. This software is only required when the design-time arguments for that component are being set. In the run-time environment, it does not need to be added.
- In the persistent storage, the configuration settings of a Bean can be stored and resumed at a later time.
- You can also use Bean to register the receive events from other objects and generate events that are sent to other objects.

Introspection

Introspection is present at the core of Java Beans. It is defined as looking up or examining a Bean to determine its functionalities. It allows another application, such as a design tool, to obtain information about a component; therefore, it is an essential feature of the Java Bean API. The Java Bean technology cannot operate without introspection.

There are two methods through which the developer can specify the properties, methods, and events needed to be explored by an application builder tool. The first method uses the naming conventions that allow ways to inspect the information about the Bean. In the second method, an additional class that extends the `BeanInfo` interface is provided, which explicitly provides this information.

Design Patterns for Properties

Design patterns are used as solutions to specific problems and so they are not the final designs. They provide developers a common language understandable at both ends. For example, if one developer understands about `builder`, then the other developer knows exactly what it is. A design pattern can be reused in multiple areas, and that is the main benefit of using it. It solves the problems that occur in various situations. A new code can be created by using a design pattern. Object-oriented design patterns represent various relationships and

interactions taking place between classes or objects without showing the final application classes or objects that are added.

Each design pattern consists of the following parts:

1. **Problem/Requirement—**To create a design pattern, the first step is to determine the requirements of the problem to solve and the mini analysis of design followed by coding to test. This is usually a common problem that will arise in more than one application.
2. **Forces—**Forces indicate the technical boundaries that direct toward the solution of a problem.
3. **Solution—**This section tells how to write the code to solve a problem. This is the design part of the design pattern. It may contain class diagrams, sequence diagrams, and others to indicate how to write the code for the solution.

A property represents a subgroup of a Bean's state. The properties of a component are assigned some values that represent its behavior and appearance. Properties can be of three types: simple, Boolean, and indexed.

Simple Properties

A simple property contains a single value. The following design patterns are used to identify it:

```
public K getName();
public void setName(K arg);
```

where Name is the name of the property and K is its type.

A read/write property contains both the get() and set() methods to access the values. A read-only property contains a `get()` method and a write-only property contains a `set()` method.

Boolean Properties

A Boolean property consists of only two values, either true or false. It can be identified by the following design patterns:

```
public boolean isN();
public boolean getN();
public void setN(boolean value);
```

where N is the name of the property.

Indexed Properties

An indexed property can have multiple values. The following design patterns are used to identify an indexed property:

```
public K getName(int index);
public void setName(int index, K value);
public void setName(K values[]);
```

where Name is the name of the property and K is its type.

We will discuss these properties in *Immediate Solutions* section in detail.

Design Patterns for Events

The delegation event model has been discussed earlier in this book and it is used by the Beans. The Beans can perform various operations, such as they can fire events and transfer them to other objects. These can be identified by the following design patterns, where `T` is the type of the event:

```
public void addTListner(TListener eventListener);
public void removeTListner(TListener eventListener);
```

To insert or delete a listener for the specified event, these methods are used. For example, assuming an event interface type called `TemperatureListener`, a Bean that examines temperature might supply the following mentioned methods:

```
public void addTemperatureListner(TemperatureListener tl) { . . . }
public void removeTemperatureListner(TemperatureListener tl) { . . . }
```

Methods and Design Patterns

If a problem occurs repeatedly, a solution needs to be implemented. That solution is called a pattern. The design patterns provide language-independent strategies to solve problems related to the object-oriented design. While making a design, the names of some common solutions must be known beforehand. Design patterns make the communication effective. Further, they are not used to name non-property methods. The introspection method finds all of the public methods of a Bean protected, while the private methods are not viewed.

Using the BeanInfo Interface

We have discussed earlier that design patterns implicitly determine the information available to the user. The role of the `BeanInfo` interface is to enable you to explicitly control what information is available. The `BeanInfo` interface defines several methods. To discuss it further, let's introduce the following methods:

```
PropertyDescriptor[] getPropertyDescriptors()
EventSetDescriptor[] getEventSetDescriptors()
MethodDescriptor[] getMethodDescriptors()
```

The role of the return array of objects is to provide information about the properties, events, and methods of a Bean. With respect to this, the classes `getPropertyDescriptors()`, `getEventSetDescriptors()`, and `getMethodDescriptors()` methods are well defined within the `java.beans` package, and these methods describe the indicated elements. By implementing these methods, a developer can designate exactly what is shown to a user, by passing introspection based on design patterns.

We will discuss these methods in *Immediate Solutions* section in detail.

Persistence

Persistence means saving the current position, properties, and instance variables of a Bean. Basically, it means restoring the current position of a Bean with the values of its properties and instance variables to nonvolatile storage. One way to serialize a Bean is to implement the `java.io.Serializable` interface, which acts like a marker interface. Implementing the `java.io.Serializable` interface makes serialization automatic.

We will discuss it in *Immediate Solutions* section in detail.

Customizers

The role of the customizer, which is provided by a Bean developer, is to help another developer configure the Bean.

A customizer provides step-by-step information about the process to be followed to use a component in a specific context.

To develop a customizer, a Bean developer has greater flexibility to differentiate his or her product in the marketplace.

Immediate Solutions

Understanding Java Beans

The Novice Programmer appears and says, "Hmm!! The user of Java seems to be a little confused about Java Bean and its relation to Java class. Moreover, they ask about the different processes related to it and how these processes function and help them." "Java has described and solved these queries quite simply. You need not worry about this. Since, a Java Bean can be any other Java class, both of these can be manipulated in a visual builder tool and composed into any application as per the requirement of the user. There are certain other advantages with regard to it", you say.

A Java Bean is any Java class that has limited conventions related to property and event interface definitions. It can be calculated in a visual builder tool and formed into applications.

Introspection is the method by which a builder tool examines how a Bean operates and also discriminates Beans from the Java classes. Beans have their predefined patterns for their method parameters and class definitions; tools that identify these patterns can view inside a Bean and specify its properties and behavior.

Moreover, introspection calculates Bean's state at design time, that is, the time it is gathered as a part from a larger application. The parameters supplied to a method within Beans must follow a specific pattern so as to help the introspection tools in identifying how Beans can be manipulated at runtime and design time.

Designing Programs Using Java Beans

"So how do I use Java Beans anyway?" the NP wants to know. "You use them in application builder tools," you say, "such as the BeanBox tool that comes with the Bean Development Kit." "Show me," the NP says.

After you have downloaded and installed the Bean Development Kit (BDK), which is currently at `http://www.ecst.csuchico.edu/~amk/foo/csci611/notes/beans/BDK/`, you can work with beans in the BeanBox tool that comes with the BDK. Assuming that you have installed the BDK (the default path varies by system), you can open the BeanBox; for example, in Windows, you use the run.bat file in the beanbox directory. When you run this batch file, the BeanBox opens, as shown in Figure 23.1.

You can see the available beans in the toolbox on the left. When you add beans to an application, they'll appear in the BeanBox window next to the toolbox. You can also use the Properties window to set the properties that have been designed into a bean, and you can use the Method Tracer window to handle method execution. Here's an example showing how to work with some beans.

Click the Juggler bean in the toolbox, which changes your cursor to a cross. Next, "draw" a Juggler bean in the BeanBox by dragging the mouse (this bean displays the Java mascot image "juggling" coffee beans). Do the same for the bean named `OurButton`. This results in the application under design (Figure 23.2.) You can connect the beans in a BeanBox, thus creating a single application from several beans. For example, you can connect the button to the Juggler bean, so when you click the button, the juggler will stop juggling.

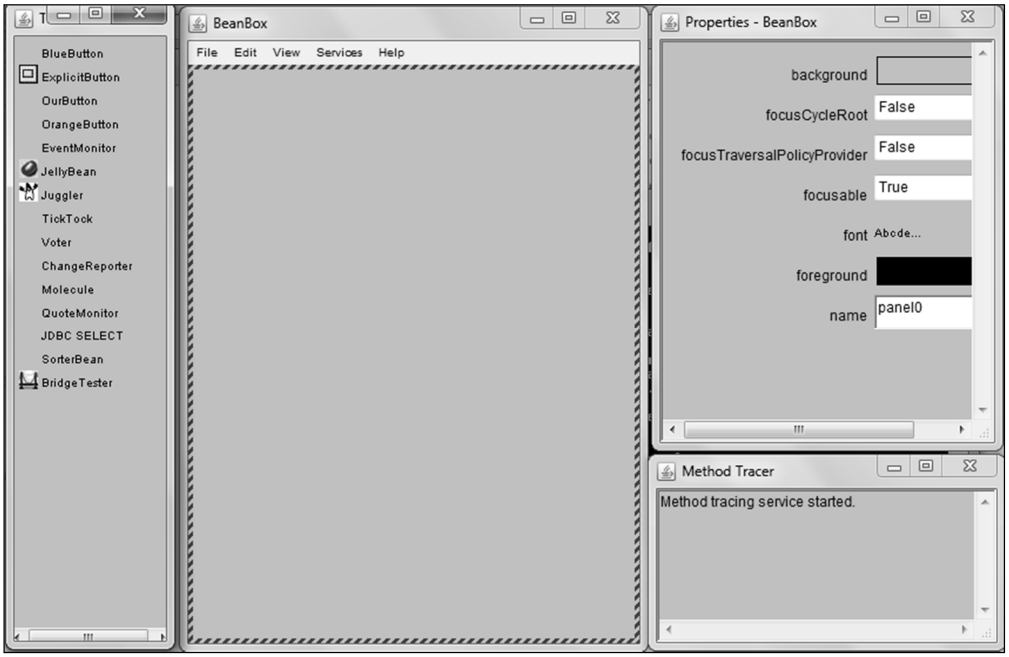


Figure 23.1: Using the BeanBox

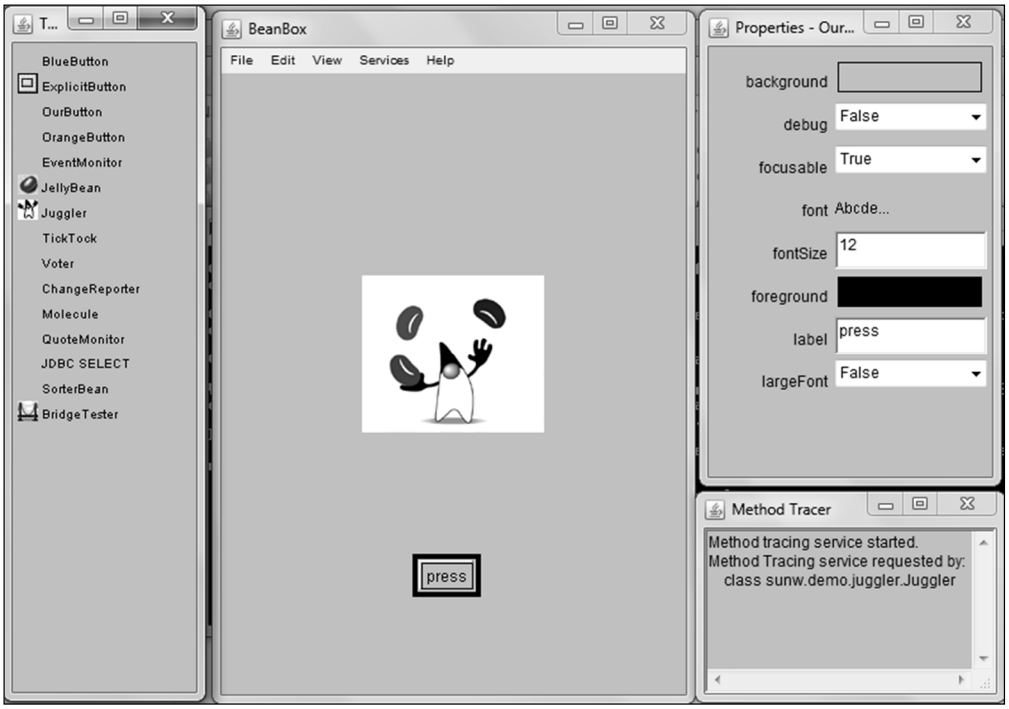


Figure 23.2: Creating an application

First, click the button bean in the BeanBox. The Properties window will display the properties you can set for this bean, including its label—if you want to enter a new label for this button (such as "Click Me"), you can do it in the Properties window (you'll see how to set the properties of your own beans this way later). To make the button do something when you click it, select the Edit | Events | action | actionPerformed menu item now. When you do, a red line appears between the mouse location and the button. Stretch that line to the Juggler bean now and click the Juggler bean, as shown in Figure 23.3. This connects the button and the Juggler bean.

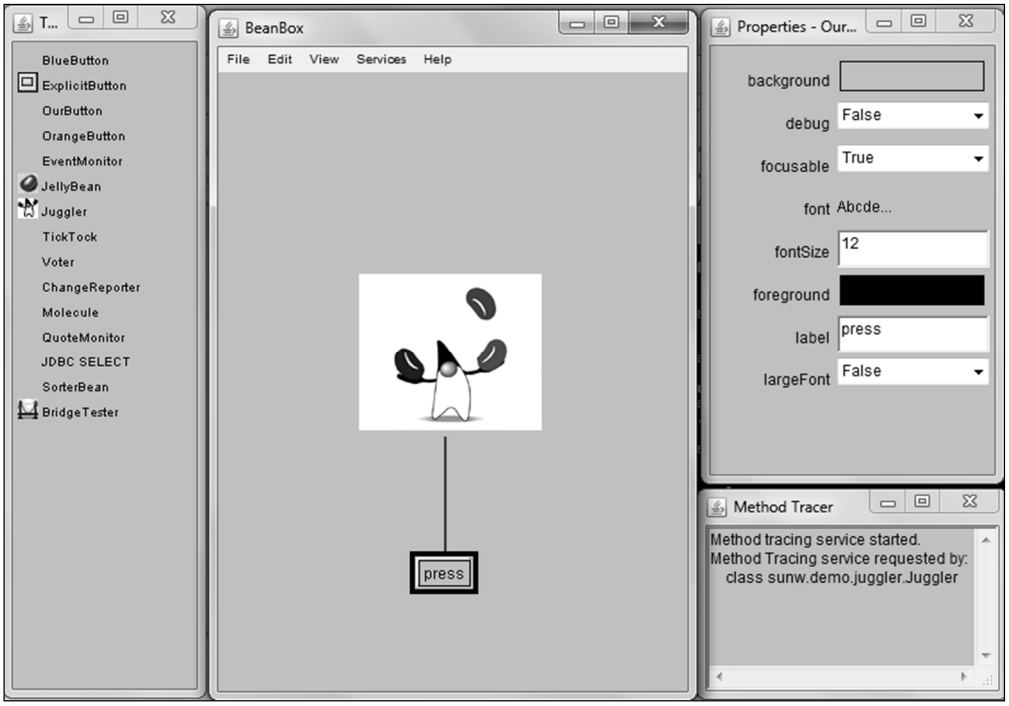


Figure 23.3: Connecting one bean to another

When you click the Juggler bean, the EventTargetDialog box appears, as shown in Figure 23.4, displaying the available methods you can call in the Juggler bean when the button is clicked:

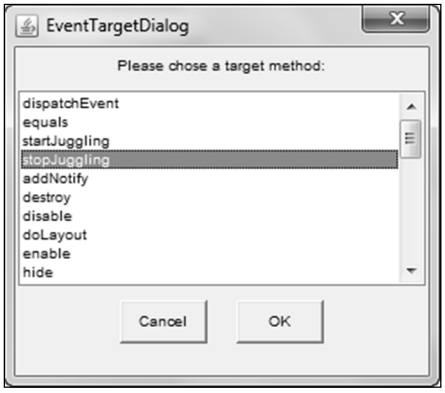


Figure 23.4: The EventTargetDialog box

For this example, choose the `stopJuggling()` method to make the juggler stop juggling; then, click the OK button to close the EventTargetDialog box.

Now, when you click the button in the BeanBox, the juggler will stop juggling. Congratulations, you have connected two beans, thus creating a new program.

How do you actually run this program outside the BeanBox? Take a look at the next solution.

Creating Applets that Use Java Beans

"OK," says the NP. "I have designed an applet with plenty of beans in it and I want to create a JAR file that I can let people use. How do I do that?" "Simple," you say. "Maybe," the NP says, "but *how*?"

After you have designed an applet using the BeanBox, you can use the `MakeApplet` item in the File menu to create a JAR file containing your new applet and all the beans it uses. When you select this menu item, it opens the Make an Applet dialog box, you see in Figure 23.5:

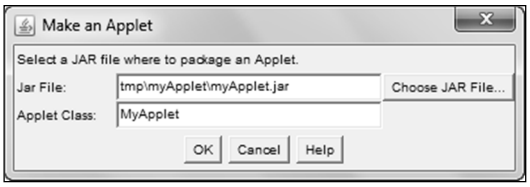


Figure 23.5: The Make an Applet dialog box

In this example, we'll use the applet we developed in the previous solution. We'll stick to the default JAR file name, `myApplet.jar`, and applet class, `MyApplet`. When we click OK, the `myApplet.jar` file is created.

NOTE

If you want to use any of the beans that come with the BDK in an applet JAR file, you'll have to compile them first and make sure the java compiler can find their class files. You'll find those beans in the demo\sunw\demo directory. In addition, you'll need to compile the AppletSupport.java file in the beanbox\sunw\beanbox directory and make sure the java compiler can find the resulting class files.

The `myApplet.jar` file can be used with Web browsers and the Oracle appletviewer. Here's the HTML page that we'll use with the `myApplet.jar file`:

```
<HTML>
   <HEAD>
         <TITLE>Beans applet example</TITLE>
   </HEAD>
   <BODY>
         <APPLET>
                CODE=MyApplet.class
                WIDTH=200
                HEIGHT=200>
                <PARAM NAME=archive VALUE="myApplet.jar">
         </APPLET>
   </BODY>
</HTML>
```

After running the code, you can see the applet in the appletviewer.

Creating a Java Bean

The NP appears and says, "I have used Java Beans now—but how do I actually create my own beans?" "Well," you say, "it's not too hard, but it does take a little work." "Uh-oh," says the NP.

We'll create a simple Java Bean in this solution to show how beans work, and we'll elaborate on this bean in the rest of the chapter. This bean will just draw itself in red, and when you click it, it'll display a count of the number of times it's been clicked.

We'll place this bean in the BDK's demo directory, so we create a directory named bean—demo\sunw\demo\bean—and store the class files for this bean in that directory. We'll start the code, bean.java, by indicating that this bean is part of a package named sunw.demo.bean:

```
package sunw.demo.bean;
import java.awt.*;
import java.awt.event.*;
. . .
```

As far as the actual bean class goes, we use the Canvas class to draw the bean itself (you can also use other classes, such as the Panel class, which we'll cover in this chapter). The rest of the code is pretty ordinary; we just add a mouse listener to the canvas to record mouse clicks and set the size of the canvas (thus setting the size of the bean):

```
package sunw.demo.bean;
import java.awt.*;
import java.awt.event.*;
public class bean extends Canvas
{
  int count;
  public bean()
  {
        addMouseListener(new MouseAdapter()
        {
                public void mousePressed(MouseEvent me) { clicked(); }
                });
                count = 0; setSize(200, 100);
        }
  . . .
}
```

Finally, we implement the method that handles mouse clicks, and we draw the bean, including the click count, in the paint() method:

```
  package sunw.demo.bean;

  import java.awt.*;
  import java.awt.event.*;

  public class bean extends Canvas
  {
        int count;
        public bean()
        {
                addMouseListener(new MouseAdapter() {
                public void mousePressed(MouseEvent me) { clicked(); }
        });
        count = 0; setSize(200, 100); }
        public void clicked() { count++; repaint(); }
        public void paint(Graphics g)
        {
                Dimension dimension = getSize();
                int height = dimension.height;
                int width = dimension.width;
                g.setColor(new Color(255, 0, 0));
                g.fillRect(0, 0, --width, --height);
                g.setColor(new Color(0, 0, 0));
                g.drawString("Click count = " + count, 50, 50);
  }
}
```

As you can see, the code for a new Java Bean is pretty simple—essentially, you just draw a component. However, it's not a bean yet—we still have to put it into a JAR file and label it in that JAR file as a bean. To learn more about how this works, see the next solution.

Creating a Bean Manifest File

The NP appears and says, "Well, I have written the code for my new Java Bean, but now what?" You smile and say, "You have to pack the bean into a JAR file and indicate that it's a bean in the JAR file's manifest file." "Tell me more!" says the NP.

To make a bean into a bean, you have to store the class file(s) to use in a JAR file, and you use a manifest to indicate which classes are beans. To show you how this works, we'll use the bean we started developing in the previous solution and create a manifest file, `bean.mft`, for it here. We'll place this manifest file in the *demo* directory to make creating the JAR file easier (see the next solution).

To indicate that a class in a JAR file is a Java Bean, you have to set its `Java-Bean` attribute to True. Here's how that works with the bean introduced in the previous solution—`bean.class`. This class file is in the `sunw.demo.bean` package, which means it'll be stored in the JAR file as `sunw/demo/bean/bean.class` (like Unix, JAR files use forward slashes as directory separators). To indicate that this class file is a bean, here's what we put in the manifest file, `bean.mft`:

```
Name: sunw/demo/bean/bean.class
Java-Bean: True
```

We'll put this manifest to work in the next solution when we create the JAR file for this bean.

Creating a Bean JAR File

The NP appears and says, "OK, I have got the class files for my new bean, and I have got the manifest file I want to use. Now what?" "Now," you say, "you are ready to use the jar tool." "You do it," the NP says. "I'll watch."

In this solution, we'll create the JAR file for the bean we have been developing over the previous two solutions. First, compile `bean.java`, which will create two class files: `bean.class` and `bean$1.class`. Copy these files to the `demo\sunw\demo\bean` directory we have created. Next, place the manifest file developed in the previous solution, `bean.mft`, into the demo directory. Now we are ready to go. Make sure you are in the demo directory and use the jar tool, like this:

```
C:\...\demo>jar cfm ..\jars\bean.jar bean.mft .
```

This creates the new JAR file for this bean, `bean.jar`, and stores it in the `demo\jars` directory, which is where the BeanBox will look for it. That's how you install a bean—by placing its JAR file in that directory. The new bean is ready to go, and it's put to work in the next solution.

Creating a New Bean

We have developed a new Java Bean over the previous few solutions and installed it in the demo\jars directory. When you open the BeanBox, you'll see this bean (which we have just called "bean") listed in the toolbox. You can draw a bean of this new type in the BeanBox.

This new bean is already active in the BeanBox, as you can see by clicking it, because it shows the number of times it's been clicked. This new Java Bean is a good start, but what about doing some more, such as adding other Java controls to it? Take a look at the next solution.

Adding Controls to Beans

"I don't want to have to reinvent the wheel," says the Novice Programmer. "Which means what?" you ask. "Well, it means that if I want a button in my new Java Bean, why do I have to draw and support it myself—why can't I just use a standard Java button?" "Of course you can," you say. The NP asks, "You can?"

You can add Java controls such as buttons to your beans—you just have to make sure you base your bean on a class that's a container, such as the `Panel` class. Here's an example in which we add a button to a bean and have the bean display the number of times it has been clicked. We start by basing this bean, which we'll call "button," on the `Panel` class and adding it to the sunw.demo.button package (which means we'll store its class files in the demo\sunw\button directory). Here's how we create the panel, size it, and add a button to it:

```
package sunw.demo.button;
```

```
import java.awt.*;
import java.awt.event.*;

public class button extends Panel implements ActionListener
{
  int count;
  Button button1;
  public button()
  {
        count = 0;
        setSize(200, 100);
        button1 = new Button("Click me");
        button1.addActionListener(this);
        add(button1);
  }
}
```

All that's left now is to make the button active by incrementing the click count the button, repainting the bean to display the click count, and creating the `paint()` method:

```
package sunw.demo.button;

import java.awt.*;
import java.awt.event.*;

public class button extends Panel implements ActionListener
{
  int count; Button button1;
  public button()
  {
        count = 0;
        setSize(200, 100);
        button1 = new Button("Click me");
        button1.addActionListener(this);
        add(button1);
  }
  public void actionPerformed(ActionEvent e)
  {
        count++;
        repaint();
  }
  public void paint(Graphics g)
  {
        Dimension dimension = getSize();
        int h = dimension.height;
        int w = dimension.width;
        g.setColor(new Color(255, 0, 0));
        g.fillRect(0, 0, w-1, h-1);
        g.setColor(new Color(0, 0, 0));
        g.drawString("Click count = " + count, 50, 50);
  }
}
```

As you can see, it's easy to add Java controls to a bean.

Giving a Bean Properties

"I notice," the NP says, "that you can set the properties of a bean in the BeanBox using the Properties window. Why can't I do that with the beans I create?" "You certainly can," you say. The NP says, "Wow!"

The properties of a bean let you configure it; setting its caption, size, color, and any other aspect of the bean for which properties are defined. Although any public data member of a bean class can be treated as a property, there's a formal procedure you should follow to inform the Java framework about the properties of your beans: implementing the `BeanInfo` interface. The fields of the `BeanInfo` interface appear in Table 23.1 and its methods in Table 23.2:

Table 23.1: Fields of the BeanInfo interface	
Field	**Does this**
`static int ICON_COLOR_16x16`	It indicates a 16x16 color icon.
`static int ICON_COLOR_32x32`	It indicates a 32x32 color icon.
`static int ICON_MONO_16x16`	It indicates a 16x16 monochrome icon.
`static int ICON_MONO_32x32`	It indicates a 32x32 monochrome icon.

Table 23.2: Methods of the BeanInfo interface	
Method	**Does this**
`BeanInfo[] getAdditionalBeanInfo()`	It allows a `BeanInfo` object to return an arbitrary collection of other `BeanInfo` objects.
`BeanDescriptor getBeanDescriptor()`	It gets the bean's bean descriptor.
`int getDefaultEventIndex()`	It gets the default event. A bean may have a default event (the event that will mostly commonly be used).
`int getDefaultPropertyIndex()`	It gets the default property index. A bean may have a default property (the property that will mostly commonly be initially chosen for update).
`EventSetDescriptor[] getEventSetDescriptors()`	It gets the bean's event set descriptors.
`Image getIcon (int iconKind)`	It gets an `image` object that can be used to represent the bean in toolboxes, toolbars, and so on.
`MethodDescriptor[] getMethodDescriptors()`	It gets the bean's method descriptors.
`PropertyDescriptor[] getPropertyDescriptors()`	It gets the bean's property descriptors.

In fact, most beans don't implement the `BeanInfo` interface directly. Instead, they extend the `SimpleBeanInfo` class, which implements `BeanInfo`. Here's the inheritance diagram for `SimpleBeanInfo`:

```
java.lang.Object
  |____java.beans.SimpleBeanInfo
```

You'll find the constructor of the `SimpleBeanInfo` class in Table 23.3 and its methods in Table 23.4:

Table 23.3: The constructor of the SimpleBeanInfo class	
Constructor	**Does this**
`SimpleBeanInfo()`	It constructs a `BeanInfo` object.

Table 23.4: Methods of the SimpleBeanInfo class	
Method	**Does this**
`BeanInfo[] getAdditionalBeanInfo()`	It is implemented to indicate that there are no other relevant `BeanInfo` objects.
`BeanDescriptor getBeanDescriptor()`	It is implemented to indicate the deny of knowledge about the class and the customizer bean.
`int getDefaultEventIndex()`	It is implemented to deny knowledge of a default event.
`int getDefaultPropertyIndex()`	It is implemented to deny knowledge of a default property.
`EventSetDescriptor[] getEventSetDescriptors()`	It is implemented to deny knowledge of event sets.

Table 23.4: Methods of the SimpleBeanInfo class	
Method	**Does this**
`Image getIcon (int iconKind)`	It is implemented to deny there are no icons available.
`MethodDescriptor[] getMethodDescriptors()`	It is implemented to deny knowledge of methods.
`PropertyDescriptor[] getPropertyDescriptors()`	It is implemented to deny knowledge of properties.
`Image loadImage (String resourceName)`	It is used to help in loading icon images.

To actually describe a property, you use the `PropertyDescriptor` class, which in turn is derived from the `FeatureDescriptor` class. Here's the inheritance diagram for the `FeatureDescriptor` class:

```
java.lang.Object
  |____java.beans.FeatureDescriptor
```

You'll find the constructor of the `FeatureDescriptor` class in Table 23.5 and its methods in Table 23.6:

Table 23.5: The constructor of the FeatureDescriptor class	
Constructor	**Does this**
`FeatureDescriptor()`	It constructs a feature descriptor

Table 23.6: Methods of the FeatureDescriptor class	
Method	**Does this**
`Enumeration<string> attributeNames()`	It gets an enumeration of the locale-independent names of this feature.
`String getDisplayName()`	It gets the localized display name of this feature.
`String getName()`	It gets the programmatic name of this feature.
`String getShortDescription()`	It gets the short description of this feature.
`Object getValue (String attributeName)`	It retrieves a named attribute with this feature.
`boolean isExpert()`	It returns True for features that are intended for expert users.
`boolean isHidden()`	It returns True for features that are intended only for tool use.
`boolean isPreferred()`	It returns True for features that are particularly important for presenting to people.
`void setDisplayName (String displayName)`	It sets the localized display name of this feature.
`void setExpert (boolean expert)`	It sets the expert flag for features that are intended for expert users.
`void setHidden (boolean hidden)`	It sets the hidden flag for features intended only for tool use.
`void setName(String name)`	It sets the name of this feature.
`void setPreferred (boolean preferred)`	It sets the preferred flag, used to identify features that are particularly important for presenting to people.
`void setShortDescription (String text)`	You can associate a short descriptive string with this feature.
`void setValue (String attributeName, Object value)`	You can associate a named attribute with this feature.

Here's the inheritance diagram for the `PropertyDescriptor` class:

```
java.lang.Object
  |____java.beans.FeatureDescriptor
        |____java.beans.PropertyDescriptor
```

You'll find the constructors of the `PropertyDescriptor` class in Table 23.7 and its methods in Table 23.8:

Table 23.7: Constructors of the PropertyDescriptor class

Constructor	Does this
`PropertyDescriptor(String propertyName, Class<?> beanClass)`	It constructs a PropertyDescriptor for a property that follows the standard Java convention by having `getFoo()` and `setFoo()` accessor methods.
`PropertyDescriptor(String propertyName, Class<?> beanClass, String readMethodName, String writeMethodName)`	It takes the name of a simple property as well as `method` names for reading and writing the property.
`PropertyDescriptor(String propertyName, Method readMethod, Method writeMethod)`	It takes the name of a simple property as well as `Method` objects for reading and writing the property.

Table 23.8: Methods of the PropertyDescriptor class

Method	Does this
`PropertyEditor createPropertyEditor(Object bean)`	It creates an instance of a property editor using the current property editor class.
`boolean equals(Object obj)`	It compares this PropertyDescriptor against the specified object.
`Class<?> getPropertyEditorClass()`	It gets any explicit `PropertyEditor` class that has been registered for this property.
`Class<?> getPropertyType()`	It gets the `Class` object for the property.
`Method getReadMethod()`	It gets the method that should be used to read the property value.
`Method getWriteMethod()`	It gets the method that should be used to write the property value.
int hashCode()	It gets a hash code value for the object.
`boolean isBound()`	It updates to bound properties and that will cause a `PropertyChange` event to be fired when the property is changed.
`boolean isConstrained()`	It attempts updates to constrained properties and that will cause a `VetoableChange` event to be fired when the property is changed.
`void setBound (boolean bound)`	It updates to bound properties and that will cause a `PropertyChange` event to be fired when the property is changed.
`void setConstrained (boolean constrained)`	It attempts updates to constrained properties and that will cause a `VetoableChange` event to be fired when the property is changed.
`void setPropertyEditorClass(Class<?> propertyEditorClass)`	Normally, property editors will be found using the property editor manager.
`void setReadMethod (Method getter)`	It sets the method that should be used to read the property value.
`void setWriteMethod (Method setter)`	It sets the method that should be used to write the property value.

Let's look at an example that implements a property in a Java Bean. In this case, we'll add a property named `filled` to the click-counting operation developed in this chapter. This property is a Boolean property that, when True, makes sure the bean will be filled in with color. We'll call this new bean `Bean2`.

To keep track of the new `filled` property, we'll add a private Boolean variable of that name to the `Bean2` class:

```
package sunw.demo.bean2;
import java.awt.*;
import java.awt.event.*;
public class Bean2 extends Canvas {
private boolean filled;
. . .
```

We initialize this property to False when the bean is created:

```
package sunw.demo.bean2;

import java.awt.*;
import java.awt.event.*;

public class Bean2 extends Canvas
{
  private boolean filled;
  int count;
  public Bean2()
  {
        addMouseListener(new MouseAdapter()
        {
              public void mousePressed(MouseEvent me) { clicked(); }
        });
        count = 0; filled = false;
        setSize(200, 100);
  }
  . . .
}
```

When you implement a property, Java will look for two methods: `get`*`PropertyName`*`()` and `set`*`PropertyName()`*, where *`PropertyName`* is the name of the property. The `get()` method returns the current value of the property, which can be any supported type, and the `set()` method takes an argument of that type, which you are supposed to set the property to. Here's how we implement the `getfilled()` and `setfilled()` methods:

```
package sunw.demo.bean2;

import java.awt.*;
import java.awt.event.*;

public class Bean2 extends Canvas
{
  private boolean filled;
  int count;
  public Bean2()
  {
        addMouseListener(new MouseAdapter() {
              public void mousePressed(MouseEvent me) { clicked(); }
        });
        count = 0;
        filled = false;
        setSize(200, 100);
  }
  public void clicked()
  {
        count++;
        repaint();
  }
  public boolean getfilled() { return filled; }
  public void setfilled(boolean flag)
  {
        this.filled = flag;
        repaint();
  }
  . . .
}
```

All that's left is to put the `filled` property to use. If it's True, the bean should be filled in with color, so here's the code we add to the `paint()` method:

```
public void paint(Graphics g)
{
  Dimension dimension = getSize();
  int height = dimension.height;
  int width = dimension.width;
  if(filled)
  {
        g.setColor(new Color(255, 0, 0));
        g.fillRect(0, 0, --width, --height);
  }
  g.setColor(new Color(0, 0, 0));
  g.drawString("Click count = " + count, 50, 50);
}
```

Now we have to create a new class, `Bean2BeanInfo`, which will return information about this new bean property. This class will go into the same package as the bean itself, but it's based on the `SimpleBeanInfo` class:

```
package sunw.demo.bean2;
import java.beans.*;
public class Bean2BeanInfo extends SimpleBeanInfo { . . . }
```

To let Java know about properties, you implement the `getPropertyDescriptors()` method, which returns an array of `PropertyDescriptor` objects. Each `PropertyDescriptor` object holds the name of a property and points to the class that supports that property. Here's how we create a `PropertyDescriptor` object for the `filled` property:

```
package sunw.demo.bean2;
import java.beans.*;
public class Bean2BeanInfo extends SimpleBeanInfo {
  public PropertyDescriptor[] getPropertyDescriptors() {
try {
PropertyDescriptor filled = new
PropertyDescriptor("filled", Bean2BeanInfo.class);
. . . }
catch(Exception e) { }
return null; }
}
```

This is the only property in `Bean2`, so we add the new `PropertyDescriptor` object to an array and return it, like this:

```
package sunw.demo.bean2;
import java.beans.*;
public class Bean2BeanInfo extends SimpleBeanInfo
{
  public PropertyDescriptor[] getPropertyDescriptors()
  {
        try {
              PropertyDescriptor filled = new
              PropertyDescriptor("filled", Bean2BeanInfo.class);
              PropertyDescriptor propertydescriptor[] = {filled};
              return propertydescriptor;
        }
        catch(Exception e) {}
        return null;
  }
}
```

After compiling this new class (Bean2BeanInfo.java), we place Bean2BeanInfo.class in the directory demo\sunw\demo\bean2, along with the classes created when we compiled Bean2.java. We'll need a new manifest file that includes the Bean2BeanInfo class. Here's what bean2.mft looks like:

```
Name: sunw/demo/bean2/Bean2.class
Java-Bean: True
Name: sunw/demo/bean2/Bean2BeanInfo.class
Java-Bean: True
```

We place this new manifest file in the demo directory. Finally, here's how we create the new bean2.jar file and install it:

```
C:\...\demo>jar cfm ..\jars\bean2.jar bean2.mft .
```

Now when we run the BeanBox and add a new bean2 bean to the BeanBox, the new filled property will appear in the Properties window. Setting filled to True causes the bean to be filled with color, which is also shown in the figure.

Design Patterns for Properties

The NP appears and says, "Well, well, I have come to know about Java Bean and its uses. However, in Java, how will you get properties? What are its design patterns? Can you enlighten me in this regard to make me doubly sure?" "Yes Sir", you say. "There are two different design patterns for properties in Java. You can even add color properties to Java Bean", you reply to his queries.

The subset of a Bean's state is a property. The behavior and appearance of that component is determined by the values assigned to the properties. The setter() method sets a property and the getter() method retrieves that property. There are three types of properties, namely, simple, Boolean, and indexed.

Using Simple Properties

The characteristic of the simple property is that it has a single value. It is specified by the following design patterns, where N is the name of the property and T is its type:

```
public T getN();
public void setN(T arg);
```

To access its values, a read/write property has both of these methods. A read-only property has only a get() method and a write-only property has only a set() method. The following are three read/write simple properties along with their getter() and setter() methods:

```
private double depth, height, width;
public double getDepth() { return depth; }
public void setDepth(double dbl) { depth = dbl; }
public double getHeight() { return height; }
public void setHeight(double hdbl) { height = hdbl; }
public double getWidth() { return width; }
public void setWidth(double wdbl) { width = wdbl; }
```

Adding a Color Property to SimpleBean

To add a Color property, you have to make the following changes to SimpleBeanapp.java:

```
Create and initialize a private instance variable.
private Color color = Color.green;
Write a public getter method.
public Color getColor() { return color; }
Write a public setter method.
public void setColor(Color newclr) {
  color = newclr;
  repaint(); }
Override the paint() method inherited from Canvas.
public void paint(Graphics g) {
```

```
    g.setColor(color);
    g.fillRect(20, 5, 20, 30);
  }
```

Compile, load in the Toolbox, and create an instance of the Bean. The following results are discussed here:

- `SimpleBean` is shown with a green-centered rectangle.
- Consequently, the Properties window will include a new color property. Simultaneously, the introspection process will also look for a color property editor. A color property editor is one of the default editors supplied with the `BeanBox class`, and is defined as `SimpleBean's` color property editor. To execute this editor, click on the color property in the Properties window.
- In simple language, Figure 23.6 shows the new `SimpleBean` instance within the `BeanBox`; `SimpleBean's` new color property within the Properties window and the color property editor loaded with the `BeanBox`.

By menu or by RGB value, you can change the color property. Therefore, you can try to change colors. The following is the complete `SimpleBeanapp` source code that has been revised to add a color property:

```
import java.awt.*;
import java.io.Serializable;
public class SimpleBeanapp extends Canvas implements Serializable {
  private Color clr = Color.green;
  public Color getColor() { return clr; }
  public void setColor(Color newclr) {
        clr = newclr;
        repaint(); }
  public void paint(Graphics g) {
        g.setColor(clr);
        g.fillRect(20, 5, 20, 30); }
  public SimpleBeanapp() {
        setSize(60,40);
        setBackground(Color.red); }
}
```

The result is as shown in Figure 23.6:

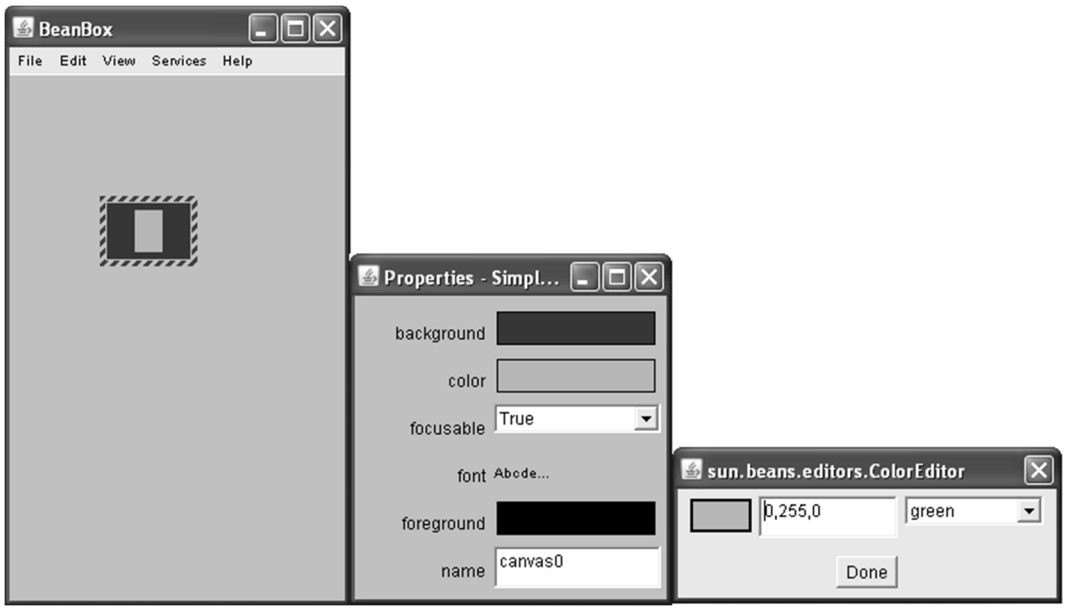


Figure 23.6: Showing SimpleBean Instance within the BeanBox

Using Indexed Properties

Indexed properties consist of multiple values. By the following design pattern, it can be identified. In this design pattern, `N` is the name of the property and `T` is its type:

```
public T getN(int index);
public void setN(int index, T value);
public T[] getN();
public void setN(T values[]);
```

Here is an indexed property called data along with the `getter()` and `setter()` methods:

```
private double data[];
  public double getData (int ind) { return data[ind]; }
  public void setData(int ind, double dblvalues) { data[ind] =  dblvalue; }
  public double [] getData() { return data; }
public void setData(double[] dblvalue) {
data = new double[dblvalue.length];
System.arraycopy(dblvalues, 0, data, 0, 0, dblvalues.length);
}
```

Using Boolean Properties

A Boolean property consists of only two values, either true or false. It can be identified by the following design patterns:

```
public boolean isN();
public boolean getN();
public void setN(boolean value);
```

where N is the name of the property.

To retrieve the values of a Boolean property, either the first or second pattern is used; if a class consists of both of its methods, then it uses the first pattern. The following example shows a class that has one Boolean property:

```
public class polygon
{
  private boolean square = false;
  public boolean isSquare( )
  {
        return square;
  }
  public void setSquare(boolean square)
  {
        this.square = square;
  }
}
```

Designing Patterns for Events

The delegation event model has been discussed earlier in this book and it is used by the Beans. The Beans operate differently, such as they can generate events and provide to other objects. These can be identified by the following design patterns, where `T` is the type of the event:

```
public void addTListner(TListener eventListener);
public void removeTListner(TListener eventListener);
```

The preceding methods help to add new events or deleting the already existing ones. For instance, consider an event interface type known as `TemperatureListener`; a Bean that monitors temperature can supply the following methods:

```
public void addTemperatureListner(TemperatureListener tl) { . . . }
public void removeTemperatureListner(TemperatureListener tl) { . . . }
```

Learning Methods and Design Patterns

For naming non-property methods, design patterns are not used. Introspection searches all of the public methods of a Bean protected, while the private methods are not available.

Using the BeanInfo Interface

As we discussed in the *In Depth* section, the `BeanInfo` interface defines several methods. To discuss it further, let's introduce the following methods:

```
PropertyDescriptor[] getPropertyDescriptors()
EventSetDescriptor[] getEventSetDescriptors()
MethodDescriptor[] getMethodDescriptors()
```

Here, the role of the return array of objects is to provide information about the properties, events, and methods of a Bean. In this regard, the classes `PropertyDescriptor`, `EventSetDescriptor`, and `MethodDescriptor` are well defined within the `java.beans` package, and these methods describe the marked elements. With the implementation of these methods, a developer can design the same what is presented to the user, bypassing introspection based on design patterns.

At the time of creating a class that implements `BeanInfo` interface, you have to remember that you must call that class `bnameBeanInfo`, where `bname` is the name of the Bean. For instance, assume that the name of the Bean is `ExampleBean`. In this case, the name of the information class will be called `ExampleBeanBeanInfo`.

Now, the question is how to simplify the use of `BeanInfo`? In this package, to simplify the `BeanInfo`, Java Beans supplies the `SimpleBeanInfo` class. It provides implementations of the `BeanInfo` interface by default, taking the three methods into consideration. However, you are free to extend this class and override one or more of the methods to explicitly control what aspects of a Bean are exposed. If you fail to override a method, then the design-pattern introspection will be used to find Bean's properties. For example, if you don't override `setPropertyDescriptors()` method, then design patterns are used to find Bean's properties.

Using Feature Descriptors

As discussed earlier, `BeanInfo` classes contain *descriptors* that accurately explain the target Bean's features. The BDK implements the following descriptor classes:

- The first type of descriptor is `FeatureDescriptor`. It is the base class for the other descriptor classes. It explains the features that are common to all descriptor types.
- The second type of descriptor is `BeanDescriptor`. It explains the target Bean's class type and name, and describes the target Bean's customizer class, if it exists.
- The third type is `PropertyDescriptor`, and it provides details on the target Bean's properties.
- The fourth descriptor is `IndexedPropertyDescriptor`. It is a subclass of `PropertyDescriptor` and discusses the target Bean's indexed properties.
- The fifth type of descriptor is `EventSetDescriptor`. The role of this type is to describe the events the target Bean fires.
- The sixth type is `MethodDescriptor`. It describes the target Bean's methods.
- The seventh type of feature descriptor is `ParameterDescriptor`. As the name indicates, it describes the method parameters.
- The `BeanInfo` interface declares methods that return arrays of the previous descriptors.

Creating Bound Properties

You can also create *bound* properties in Java Beans. Bound properties generate an event when their values change. This event is of type `PropertyChangeEvent` and is sent to all registered event listeners of this type. To make a property a bound property, use the `setBound()` method like this (note that we are adapting code from the previous solution):

```
package sunw.demo.bean2;
import java.beans.*;
```

```
public class Bean2BeanInfo extends SimpleBeanInfo
{
  public PropertyDescriptor[] getPropertyDescriptors()
  {
        try {
                PropertyDescriptor filled = new
                PropertyDescriptor("filled", Bean2.class);
                filled.setBound(true);

                PropertyDescriptor propertydescriptor[] = {filled};
                return propertydescriptor;
        }
        catch(Exception e) {}
        return null;
  }
}
```

Giving a Bean Methods

"Can you give a Java Bean method that other beans can call?" "Sure can," you say. "Pull up a chair and we'll go through it." "I'll get some coffee," the NP says while running off.

You can formally describe methods of Java Beans to the Java framework using the `MethodDescriptor` class. Here's the inheritance diagram of this class:

```
java.lang.Object
  |____java.beans.FeatureDescriptor
        |____java.beans.MethodDescriptor
```

You'll find the constructors of the `MethodDescriptor` class in Table 23.9 and its methods in Table 23.10:

Table 23.9: Constructors of the MethodDescriptor class

Constructor	Does this
`MethodDescriptor (Method method)`	It constructs a method descriptor.
`MethodDescriptor (Method method, ParameterDescriptor[] parameterDescriptors)`	It constructs a method descriptor from a method, providing descriptive information for each of the method's parameters.

Table 23.10: Methods of the MethodDescriptor class

Methods	Does this
`Method getMethod()`	It gets the method that this method descriptor encapsulates.
`ParameterDescriptor[]getParameterDescriptors()`	It gets the parameter descriptor for each of the parameters of this method descriptor's methods.

Many beans, however, don't bother to use `MethodDescriptor` objects, because any public bean method is accessible from other beans. Here's an example in which we create a new bean based on the ones we have developed in this chapter. This bean, `Bean3`, will count the number of times it's been clicked and will also support a method named `increment()` that, when invoked, will increment the click count.

Here's the code for `Bean3`, including the public `increment()` method:

```
package sunw.demo.bean3;
import java.awt.*;
import java.awt.event.*;
public class Bean3 extends Canvas {
int count;
public Bean3() {
  addMouseListener(new MouseAdapter() {
```

```
      public void mousePressed(MouseEvent me) { clicked(); }
      });
      count = 0;
      setSize(200, 100); }
            public void clicked() {
   count++;
   repaint(); }
            public void increment() {
            count++;
            repaint(); }
            public void paint(Graphics g) {
            Dimension dimension = getSize();
            int height = dimension.height;
            int width = dimension.width;
                    g.setColor(new Color(255, 0, 0));
                    g.fillRect(0, 0, --width, --height);
            g.setColor(new Color(0, 0, 0));
            g.drawString("Click count = " + count, 50, 50); }
   }
```

After creating `Bean3` and adding it to the BeanBox, we can connect other beans to the `increment()` method. We have connected a button to that method, and each time the button is clicked, the click count in the `Bean3` bean is incremented and displayed.

Giving a Bean an Icon

You may have noticed that some beans have icons in the BeanBox. You can add your own icons to your beans—all you have to do is add a `getIcon()` method to the `BeanInfo` class. Here's how to implement this method and handle all possibilities — monochrome or color icons of either 16×16 or 32×32 pixels:

```
    public java.awt.Image getIcon(int iconKind) {
    if (iconKind == BeanInfo.ICON_MONO_16x16 ||
    iconKind == BeanInfo.ICON_COLOR_16x16 ) {
    java.awt.Image image = loadImage("Icon16.gif");
    return image; }
    if (iconKind == BeanInfo.ICON_MONO_32x32 ||
    iconKind == BeanInfo.ICON_COLOR_32x32 ) {
    java.awt.Image image = loadImage("Icon32.gif");
    return image; }
    return null;
    }
```

Creating a **BeanInfo** Class

This part of the chapter uses the `ExplicitButtonBeanInfo` demo class to explain the creation of the `BeanInfo` class. There are different steps for creating a `BeanInfo` class:

First name your `BeanInfo` class. As a naming convention, you must add "`BeanInfo`" to the target class name. For example, if the target class name is `CreateButton`, then its bean information class name should be `CreateButtonBeanInfo`. Then make a `BeanInfo` class by extending the SimpleBeanInfo class as follows:

```
    public class ExplicitButtonBeanInfo extends SimpleBeanInfo {
```

By extending the `SimpleBeanInfo` class, you only have to override those methods you need, not all the methods of the `SimpleBeanInfo` class. So, override those methods which return the properties, methods, or events that you need. In the following code, the `ExplicitButtonBeanInfo` demo class overrides the `getPropertyDescriptors()` method to return four properties:

```
    public PropertyDescriptor[] getPropertyDescriptors()
    {
      try {
            PropertyDescriptor background =
            new PropertyDescriptor("background", beanClass);
```

```
            PropertyDescriptor foreground =
            new PropertyDescriptor("foreground", beanClass);
            PropertyDescriptor font =
            new PropertyDescriptor("font", beanClass);
            PropertyDescriptor label =
            new PropertyDescriptor("label", beanClass);
            background.setBound(true);
            foreground.setBound(true);
            font.setBound(true);
            label.setBound(true);
            PropertyDescriptor rv[] =
            {background, foreground, font, label};
            return rv; }
      catch (IntrospectionException e) { throw new Error(e.toString()); }
    }
```

If overriding is not done on the `SimpleBeanInfo` default method, which returns null, low-level reflection is used for that characteristic. After overriding the appropriate method, associate an icon with the target Bean as follows:

```
    public java.awt.Image getIcon(int icontype)
    {
      if (icontype == BeanInfo.ICON_MONO_16x16 ||
      icontype == BeanInfo.ICON_COLOR_16x16 ) {
            java.awt.Image img = loadImage("tom.gif");
            return img; }
      if (icontype == BeanInfo.ICON_MONO_32x32 ||
      icontype == BeanInfo.ICON_COLOR_32x32 ) {
            java.awt.Image img = loadImage("ExplicitButtonIcon32.gif");
            return img; }
            return null;
    }
```

The `BeanBox` then presents this icon next to the Bean name in the `ToolBox`. You can expect the builder tools to do the same. Specify the target `Bean` class, and if the Bean has a customizer, specify it too as follows:

```
    public BeanDescriptor getBeanDescriptor() {
      return new BeanDescriptor(beanClass, customizerClass);
    }
    . . .
    resulting,private final static Class beanClass = ExplicitButton.class;
    private final static Class customizerClass = OurButtonCustomizer.class;
```

In the previous steps, locate the `BeanInfo` class in the same directory as its target class. Firstly, the `BeanBox` searches for a target Bean's `BeanInfo` class in the target Bean's package path. If no `BeanInfo` is found, then secondly the Bean information package searches path (maintained by the `Introspector`). The default Bean information search path is `sun.beans.infos`. Again, if no `BeanInfo` class is found, then low-level reflection is used to discover a Bean's features.

Setting Bound and Constrained Properties

"Hmm!!" says the NP, "Users have some complaints about Java Bean. They are fine with the design patterns for properties. But what about the different variations on standard properties? Aren't that a bit confusing?" You say with a big smile, "Not at all, sir. It is not confusing but very simple. Java Bean supports two variations on standard properties and they are very helpful to the user of Java Bean."

A Bean that contains a bound property generates an event when the property is changed. Here, the event is of type `PropertyChangeEvent`. And, it is sent to objects that previously registered an interest in receiving such notifications. A class that handles this event must implement the `PropertyChangeListener` interface.

The Java Beans model supports two variations on standard properties—the first one is called bound properties and the second one is called constrained properties.

The task of the bound properties is to provide the registration and notification of parties whenever there is modification in the values. The constrained properties take this notification model to a step ahead by allowing the notified party to perform a veto in order to prevent the property change. Different from event handling, Java Beans handle the bound and constrained properties.

To whom are the Bound properties useful? Bound properties are useful for Beans that want to allow instances of the same Bean class or some other Bean class to monitor a property value and subsequently change their values accordingly (i.e. to match the "trend setting" Bean). Let's cite an example. Consider a GUI Bean that wants to allow other GUI Beans to keep a check on change in background color to update their backgrounds accordingly.

However, note that implementation of a `bound` property is quite easy because of the user-friendly underlying framework provided by the Java Beans architecture. `Bean` class supports a bound property. In order to provide a `bound` property, the `Bean` class must instantiate an object in the Java Beans framework, which provides a lot of this functionality, and implements registration and unregistration methods that simply invoke the appropriate methods in the Java Beans framework:

```
private PropertyChangeSupport changes = new PropertyChangeSupport(this);
public void addPropertyChangeListener(PropertyChangeListener p) {
  changes.addPropertyChangeListener(p);
}
public void removePropertyChangeListener(PropertyChangeListener p) {
  changes.removePropertyChangeListener(p);
}
```

Each bound property must then invoke the `firePropertyChange()` method from its `set()` method:

```
public void setMood(int mood) {
  int old = this.mood;
  this.mood = mood;
  repaint();
  changes.firePropertyChange("mood",
  new Integer(old), new Integer(mood));
}
```

At this level, the `PropertyChangeSupport` object handles the notification of all registered targets. You have to keep in mind that the `PropertyChangeSupport` object provides a general-purpose functionality following a prescribed protocol. Particularly, the method invocation for `firePropertyChange()` must provide the property name as well as the old and new values, which are passed along to notified targets.

Moreover, the listener (i.e., target object) must provide a `propertyChange()` method to receive the property-related notifications:

```
public void propertyChange(PropertyChangeEvent e) { }  // . . .
```

Here, constrained properties add the functionality, which might be the notified listener that can object to the property change and execute a veto. Additionally, the `Bean` class must instantiate the Java Beans object that provides this service, namely, `VetoableChangeSupport`, and executes the corresponding registration-related methods:

```
private VetoableChangeSupport veto = new VetoableChangeSupport(this);
public void addVetoableChangeListener(VetoableChangeListener vetolisten) {
  veto.addVetoableChangeListener(vetolisten);
}
public void removeVetoableChangeListener(VetoableChangeListener vetolisten) {
  veto.removeVetoableChangeListener(vetolisten);
}
```

Keep in mind that a Bean could provide one or more bound and one or more constrained properties. In this case, it must instantiate both `PropertyChangeSupport` and `VetoableChangeSupport` and provide both sets of registration-related methods. Also, constrained properties tend to be bound, but that is not an absolute requirement. The `set()` method for bound-constrained properties is slightly more complicated:

```
public void setMood(int md)
throws PropertyVetoException {
  vetoes.fireVetoableChange("mood",
  new Integer(this.md), new Integer(md));
```

```
        int old = this.md;
        this.md = md;
        repaint();
        changes.firePropertyChange("mood",
        new Integer(old), new Integer(md));
    }
```

In particular, the `set()` method must accommodate the exception `PropertyVetoException`. In this regard, the sequence of operations is as follows:

- Fire the vetoable change notification.
- Update the appropriate state variables.
- Fire the standard property change notification, if bound.

A veto-interested target object must implement the `vetoableChange()` method:

```
    public void vetoableChange(PropertyChangeEvent e)
    throws PropertyVetoException { . . . }
```

It exercises a veto by the following exceptions:

- Including a throws clause for `PropertyVetoException` and
- Raising the exception (throw new `PropertyVetoException()`), as appropriate.

Here, the Java Beans framework receives the propagated exception. In return, the Java Beans framework transfers it to `setMood()` through the `fireVetoableChange()` invocation. Note that `setMood()` is organized such that in the event of a veto: (1) it does not handle the exception (instead, returning and transferring the exception) and (2) it does not complete the update to state variables or the property notification for bound-property listeners.

Now, it is very much clear that the Java Beans framework supports a lot of property-related functionality and requires minimal work from the Bean implementation.

Implementing Persistence

As we have discussed in the *In Depth* section persistence is the process of restoring the current state of a Bean and its properties with its instance variables also being restored at later stage. What helps to provide persistence to Beans? The object serialization capabilities provided by the Java class libraries provide persistence to Beans.

Among the different methods available, the easiest way to serialize a Bean is to have it implement the `java.io.Serializable` interface, which is nothing but a marker interface. Implementing `java.io.Serializable` makes serialization automatic. Your Bean need not take any other action in this regard. Moreover, remember that automatic serialization can also be inherited. So, if any superclass of a Bean executes `java.io.Serializable`, then automatic serialization is achieved. But, the only restriction is that any class that implements `java.io.Serializable` must supply a parameter-less constructor.

When the user is using automatic serialization, they can optionally avoid a field from being saved by the use of the transient keyword. As a result, data members of a Bean specified as transient will not be serialized.

If a Bean does not implement `java.io.Serializable`, then we must provide serialization ourselves. This can be done by implementing `java.io.Externalizable`. Else, the configuration of your component cannot be saved by containers.

Using the Java Beans API

The NP appears and says, "Yes, I am now convinced that Java Bean is really interesting for the users. But, I still have some confusion regarding its functionality. Can you give me a nice and elaborate answer to it?" "Why not, sir!" you say. The Java Bean program has many different and useful functionalities. To make it more user friendly, it is provided with a set of classes and interfaces in the package.

Table 23.11 lists the interfaces in `java.beans` and provides description of their functionality, whereas Table 23.12 lists the classes in `java.beans`:

Table 23.11: The interfaces of java.beans

Interface	Description
`AppletInitializer`	This interface is used to initialize the Applet with Java Bean.
`BeanInfo`	Any class that implements the `BeanInfo` interface provides explicit information about the methods, properties, events, etc. of their bean.
`Customizer`	This interface is used to customize a target Java Bean.
`DesignMode`	This interface is used to define the notation of design time as a mode in which `JavaBean` instances should function.
`ExceptionListener`	This `exception` is thrown during any problem in listening.
`PropertyChangeListener`	This interface is used to change the property of listener.
`PropertyEditor`	This interface provides the methods that is used to edit the property.
`VetoableChangeListener`	This interface is used to notify the any changes made in constrained property.
`Visibility`	This interface determines that bean needs GUI or not and whether GUI is available.

Table 23.12: The classes of java.beans

Class	Description
`BeanDescriptor`	It provides the complete information about the bean.
`Beans`	Basic bean methods.
`DefaultPersistenceDelegate`	It is a concrete implementation of the abstract `PersistenceDelegate` class and is the delegate used by default for classes about which no information is available.
`Encoder`	Its methods are used to encode the Java Beans.
`EventHandler`	The `EventHandler` class is used by developers to make connections between user interface bean (source) and an application logic bean (target).
`EventSetDescriptor`	It describe a collection of events that delivered as method calls on an event listener interface.
`Expression`	It is the subclass of java.`beans`.Statement and returns the value when expression is evaluated.
`FeatureDescriptor`	It is the superclass of many descriptor classes like `PropertyDescriptor`, `EventSetDescriptor`, and `MethodDescriptor`.
`IndexedPropertyChangeEvent`	It gets delivered whenever a component of Java Beans changes a bound indexed property.
`IndexedPropertyDescriptor`	It describes a property that acts like an array having index read or write method of the array.
`Introspector`	It provides a standard way for tools to learn about the properties, events, and methods supported by a target Java Bean.
`MethodDescriptor`	It describes a particular method that a Java Bean supports for external access from other components
`ParameterDescriptor`	It allows bean implementers to provide additional information on each of their parameters.
`PersistenceDelegate`	It provides the information about the state of an instance of a given class in terms of the methods in the class's public API

Table 23.12: The classes of java.beans	
Class	**Description**
`PropertyChangeEvent`	It delivered whenever a bean changes a "bound" or "constrained" property.
`PropertyChangeListenerProxy`	It extends the `EventListenerProxy`, specifically for adding a named `PropertyChangeListener`.
`PropertyChangeSupport`	It is a utility class that can be used by beans that support bound properties.
`PropertyDescriptor`	It describes one property that a Java Bean get through a pair of accessor methods.
`PropertyEditorManager`	It `PropertyEditorManager` used to get a property editor for any given type name.
`PropertyEditorSupport`	This is a support class which helps to build property editors
`SimpleBeanInfo`	It is a support class to make it easier for people to provide the `BeanInfo` classes.
`Statement`	It represents a primitive statement in which a single method is applied to a target and a set of arguments - as in "`a.setFoo(b)`".
`VetoableChangeListenerProxy`	A class which extends the `EventListenerProxy`, specifically for associating a `VetoableChangeListener` with a "constrained" property
`VetoableChangeSupport`	It is a utility class that can be used by beans that support constrained properties.
`XMLDecoder`	It is used to read XML documents created using the `XMLEncoder` and is used just like the `ObjectInputStream`.
`XMLEncoder`	It is an alternative class, like `ObjectOutputStream`, which is used to generate a textual representation of a Java Bean.

Learning the Basics of an Event

"What is the use of Java event model in Java Bean?" The Big Boss continues, "I heard something about event notification process in Java Bean with regard to its events. What is that?" You say, "I can explain it to you. Java Beans uses Java event model to communicate. It's simple and gives different alternatives to the Java Bean user for any type of communication."

Java event model is used by Java Beans to make communication happen. Here, events provide second option to (synchronous) method invocations for any type of communication between components in which "background notifications" are appropriate. That means, here, components provide one or more computational services which can give feedback and handle other services on an event-driven, or asynchronous, or "logical interrupt" basis.

Let's cite an example: Consider a software component that principally provides telecommunications data analysis. As far as this software component is concerned, you can design this to "listen" for incoming data notifications, by adding new satellite data to its model on an asynchronous basis. In this case, the data processing component is nothing but a target of some other software component that declares the availability of data.

In an event-driven program, the source and target orientation is a matter of concern. A component can be a source for one type of event and, at the same time, can be a target for another. While working with Java Beans, some type of source functionality is required for some specific events such as temperature changes, progress-bar state changes, and so on and so forth. In general, event-driven designs are ideal for the following tasks:

- Handing user interface events:
  - Mouse actions
  - Keyboard events

- Managing/reporting inter-client connections:
  - JDBC Bean that connects to database server
  - Accepts database requests and notifies a client when the data is available
- Other events:
  - Property changes in a Bean
  - Any general-purpose notification from one object to another

The event time occurrence is followed by some type of event notification (as defined by Java Beans Component). The process for event notification delivers data related to event to each registered target object within an event object. `java.util.EventObject` is the base class for event objects. Here, new subclasses of `EventObject` specialize `java.util.EventObject` for Bean-specific event types.

Moreover, you have to note that event objects carry pertinent/timely data to registered targets; this data is typically supplied as read-only. Therefore, the target object uses the event object's access methods to retrieve data.

Using the Java Beans Conventions

The NP asks, "If I am working with a Java Bean package and I want the properties to be created or events to be distributed, can your Java Bean have facilities for it?" "Why not, it's so easy. You just have to use the Java Bean convention. That's all," you say.

Java Beans follow a simple yet very powerful naming convention. Here are the two most important Java Beans naming conventions:

- Properties are created using the `set*` and `get*` methods.
- Events are distributed using the `add*Listener` event and the `Interface*Listener` event, which is the standard Java event delegation model.

Using the Remote Notification and Distributed Notification

One of the very important characteristics of Java Bean components is that they run in the same virtual machine as the container or application that contains it. Java Beans can access remote machines/object/beans through:

- Standard socket-based client/server
- `Java RMI` — We expect that the use of Java RMI will be a general way for writing Java Beans within its networked systems.
- `JDBC` — It allows Java code, hence Java Beans, access SQL databases

Using Beans with JSP

"Hmm!" The NP appears and says, "Can the user of a Java Bean package reuse Java Beans? Can I compose it once again into my application?" You say, "Unlike other packages, Java Bean can be reused. That's because Java Bean components are Java classes that can easily be reused," you say.

TIP

Remember that any Java class can be a Java Beans component if it follows specific design conventions.

The role of JavaServer Pages technology is also to provide direct supports using Java Beans components with JSP language elements. Here, you can easily build and initialize beans and get and set the values of their properties.

Summary

In this chapter, you have learned about beans, using beans, adding control to beans, giving a bean properties, simple properties, and design pattern for properties. Then, you have learned designing pattern and methods for events. We also explained how to create bound properties, giving a bean methods, and give a bean an icon. Then, we explored the `BeanInfo` class, set bound & constrained properties, and implement persistence. Toward the end, we taught the basics of an event, Java Beans convention, use of remote notification and distributed notification, and use of beans with JSP.

In the next chapter, we will introduce you with JDBC.



24

Introducing JDBC

If you need an immediate solution to:	*See page:*
Learning about the Driver Interface	892
Learning about the DriverManager Class	892
The Connection Interface	893
The Statement Interface	896
The PreparedStatement Interface	899
Using the PreparedStatement Interface	902
The ResultSet Interface	904
Using ResultSet	907
The ResultSetMetaData Interface	908
Using the ResultSetMetaData Interface	910
Implementing JDBC Processes with java.sql Package	911
Processing the ResultSet	914
Closing the Statement	923
Closing the Connection	923
Using a Loop to Set Values in PreparedStatement Object	924
Using the Interactive SQL Tool	926

In Depth

Enterprise applications that are created using the Java EE technology need to interact with databases to store application-specific information. For example, search engines use databases to store information about the Web pages and job portals use databases to store information about the candidates and employers who access the Web sites to search and advertise jobs on the Internet. Interacting with database requires database connectivity, which can be achieved by using the Open Database Connectivity (ODBC) driver. This driver is used with Java Database Connectivity (JDBC) to interact with various types of databases, such as Oracle, MS Access, My SQL, and SQL Server. JDBC is an Application Programming Interface (API), which is used in Java programming to interact with databases. JDBC works with different database drivers to connect to different databases.

This chapter focuses on JDBC, which is used to provide database connectivity to enterprise applications. In this chapter, you first learn about JDBC drivers as well as the features of JDBC. You then learn about the JDBC APIs that provide various classes and interfaces to develop a JDBC application. Next, the use of java.sql and javax.sql packages in JDBC implementation is described in detail. Toward the end, you learn to work with transactions in the JDBC application.

What does JDBC Do?

JDBC™ is a specification from Oracle which provides a standard abstraction (API/protocol) for Java applications to communicate with different databases. It is used to write programs required to access databases. JDBC, along with the database driver, is capable of accessing databases and spreadsheets. JDBC can also be defined as a platform-independent interface between a relational database and the Java programming language. The enterprise data stored in a relational database can be accessed with the help of JDBC APIs. The JDBC API allows Java programs to execute SQL statements and retrieve results. The classes and interfaces of JDBC allow a Java application to send requests made by users to the specified Database Management System (DBMS). Instead of allowing the drivers to target a specific database, the users can specify the name of the database used to retrieve the data.

Components of JDBC

JDBC has four main components through which it can communicate with a database. These components are as follows:

- **The JDBC API—**Provides various methods and interfaces for easy and effective communication with the databases. It also provides a standard to connect a database to a client application. The application-specific user processes the SQL commands according to their need and retrieves the result in the ResultSet object. The JDBC API provides two main packages, `java.sql` and `javax.sql`, to interact with databases. These packages contain the Java SE and Java EE platforms, which conform to the Write Once Run Anywhere (WORA) capabilities of Java.
- **The JDBC DriverManager—**Loads database-specific drivers in an application to establish a connection with the database. It is also used to select the most appropriate database-specific driver from the previously loaded drivers when a new connection to the database is established. In addition, it is used to make database-specific calls to the database to process the user requests.
- **The JDBC test suite—**Evaluates the JDBC driver for its compatibility with Java EE. The JDBC test suite is used to test the operations being performed by JDBC drivers.
- **The JDBC-ODBC bridge—**Connects database drivers to the database. This bridge translates JDBC method calls to ODBC function calls, and is used to implement JDBC for any database for which an ODBC driver is available. The bridge for an application can be availed by importing the `sun.jdbc.odbc` package, which contains a native library to access the ODBC features. The JDBC-ODBC bridge driver has been removed from JDBC 4.2 API.

JDBC Specification

With the emergence of JDBC 4.2, various changes, such as support for the REF_CURSOR data type and various other features, have been introduced in JDBC API.

The specifications that are available in different versions of JDBC are as follows:

- **JDBC 1.0**—Provides basic functionality of JDBC.
- **JDBC 2.0**—Provides JDBC API in two sections: the JDBC 2.0 Core API and the JDBC 2.0 Optional Package API.
- **JDBC 3.0**—Provides classes and interfaces in two Java packages: java.sql and javax.sql. JDBC 3.0 is a combination of the JDBC 2.1 core API and the JDBC 2.0 Optional Package API. The JDBC 3.0 specification provides performance optimization features and improves the features of connection pooling and statement.
- **JDBC 4.0**—Provides the following advanced features:
  - Auto loading of the Driver interface
  - Connection management
  - `ROWID` data type support
  - Annotation in SQL queries
  - National Character Set Conversion Support
  - Enhancement to exception handling
  - Enhanced support for large objects
- **JDBC 4.1**—Introduces the following features:
  - It provides try-with-resources statement, which has the ability to close resources of type Connection, ResultSet, and Statement automatically.
  - It has also introduced the RowSetFactory interface and the RowSetProvider class, which allow building of all types of row sets that are supported by your JDBC driver.
- **JDBC 4.2**—Introduces the following features:
  - Provides support for REF_CURSOR data type that is supported by various databases
  - Includes the java.sql.DriverAction interface
  - Includes the java.sql.SQLType interface
  - Includes the java.sql.JDBCType enum
  - Includes support for large update count by returing a long value for it

JDBC Architecture

A JDBC driver is required to process the SQL requests and generate results. JDBC API provides classes and interfaces to handle database-specific calls from users. Some of the important classes and interfaces defined in JDBC API are as follows:

- DriverManager
- Driver
- Connection
- Statement
- PreparedStatement
- CallableStatement
- ResultSet
- DatabaseMetaData
- ResultSetMetaData
- SqlData

- Blob
- Clob
- DriverAction
- SQLType
- JDBCType enum

In the JDBC architecture, the `DriverManager` class plays a significant role and is present in the JDBC API. It uses some database-specific drivers to effectively connect enterprise applications to databases.

Figure 24.1 demonstrates the simple JDBC architecture:

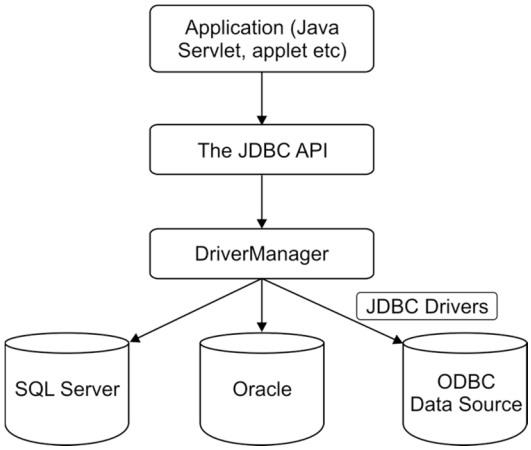


Figure 24.1: Displaying the architecture of JDBC

As shown in Figure 24.1, the Java application that needs to communicate with a database has to be programmed using JDBC API. The JDBC driver (third-party vendor implementation) supporting data source, such as Oracle, and SQL Server, has to be added in the Java application for JDBC support, which can be done dynamically at run time. The dynamic plugging of the JDBC drivers ensures that the Java application is vendor independent. In other words, if you want to communicate with any data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source. Some of the available drivers are pure Java drivers and are portable for all the environments; whereas, others are partial Java drivers and require some libraries to communicate with the database. You need to understand the architectures of all the four types of drivers (shown in Table 24.12) to decide which driver to use to communicate with the data source.

The JDBC Package

JDBC API is a part of the JDBC specification and provides a standard abstraction to use JDBC drivers. The JDBC API provides classes and interfaces that are used by Java applications to communicate to databases. The JDBC driver communicates with a relational database for any requests made by a Java application by using the JDBC API. The JDBC driver not only processes the SQL commands but also sends back the result of processing of these SQL commands. In addition, the JDBC API can be used to access the required data from all the database types, such as SQL Server, Sybase, and Oracle. A programmer does not need to write different programs to access the data from the database. The JDBC API satisfies the WORA behavior of Java. Therefore, JDBC is used largely to access data from various data sources.

The JDBC API is based upon the X/open Call Level Interface (CLI) specification and SQL standard statements. This is also the basic standard for ODBC.

The JDBC API specification is used to process and access data sources by using Java. The API includes drivers to be installed to access the different data sources. The API is used with SQL statements to read and write data from any data source in a tabular format. This facility to access data from the database is available through the group of javax.sql.RowSet interfaces. JDBC API is mainly divided into the following two packages:

- java.sql
- javax.sql

The JDBC API includes both the java.sql package, called the JDBC core API, and the javax.sql package, called the JDBC Optional Package API. Let's discuss them in detail in the following subsections.

The *java.sql* Package

The `java.sql` package is also known as the JDBC core API. This package includes the interfaces and classes for performing JDBC core operations, such as creating and executing SQL queries. The `java.sql` package consists of the interfaces and classes that need to be implemented in an application to access a database. The developer uses these operations to access the database in an application. The classes and interfaces in the `java.sql` package can be classified into the following categories based on different operations:

- Connection management
- Database access
- Data types
- Database metadata
- Exceptions and warnings

Let's discuss in brief about these categories.

Connection Management

The connection management category contains the classes and interfaces used to establish a connection with a database.

Table 24.1 describes the classes and interfaces of the connection management category:

Table 24.1: Classes and interfaces of connection management

Class/Interface	Description
`java.sql.Connection interface`	Creates a connection with a specific database. You can use SQL statements to retrieve the desired results within the context of a connection.
`java.sql.Driver interface`	Creates and registers an instance of a driver with the DriverManager object.
`java.sql.DriverManager class`	Provides the functionality to manage database drivers.
`java.sql.DriverPropertyInfo class`	Retrieves the properties required to obtain a connection.
`java.sql.SQLPermission class`	Sets up logging stream with the DriverManager object.

Database Access

SQL queries are executed to access the application-specific data after a connection is established with a database. The interfaces listed in Table 24.2 allow you to send SQL statements to the database for execution and read the results from the respective database:

Table 24.2: Interfaces of the database access category

Interface	Description
`java.sql.CallableStatement`	Executes stored procedures.
`java.sql.PreparedStatement`	Allows the programmer to create parameterized SQL statements.
`java.sql.ResultSet`	Abstracts the results of executing the `SELECT` statements. This interface provides methods to access the results row by row.
`java.sql.Statement`	Executes SQL statements over the underlying connection and access the results.
`java.sql.Savepoint`	Specifies a `Savepoint` in a transaction.

The java.sql.PreparedStatement and java.sql.CallableStatement interfaces extend the java.sql.Statement interface.

Data Types

In the JDBC API, various interfaces and classes are defined to hold specific types of data to be stored in a database. For example, to store the BLOB type values, the Blob interface is declared in the java.sql package.

Table 24.3 describes the classes and interfaces of various data types in the java.sql package:

Table 24.3: Classes and interfaces for data types in the java.sql package

Class/Interface	Description
java.sql.Array interface	Provides mapping for ARRAY of a collection.
java.sql.Blob interface	Provides mapping for the BLOB SQL type.
java.sql.Clob interface	Provides mapping for the CLOB SQL type.
java.sql.Date class	Provides mapping for the SQL type DATE. Although the java.util.Date class provides a general-purpose representation of date, the java.sql.Date class is preferable for representing dates in database-centric applications, as the type maps directly to the SQL DATE type. Note that the java.sql.Date class extends the java.util.Date class.
java.sql.NClob interface	Provides mapping of the Java language and the National Character Large Object types. The NClob interface allows you to store the values of the character string up to the maximum length.
java.sql.Ref interface	Provides mapping for SQL type REF.
java.sql.RowId interface	Provides mapping for Java with the SQL ROWID value.
java.sql.Struct interface	Provides mapping for the SQL structured types.
java.sql.SQLXML interface	Provides mapping for the SQL XML types available in the JDBC API.
java.sql.Time class	Provides mapping for the SQL type TIME and extends the java.util.Date class.
java.sql.Timestamp class	Provides mapping for the SQL type TIMESTAMP and extends the java.util.Date class.
java.sql.Types class	Holds a set of constant integers, each corresponding to a SQL type.

The JDBC API provides certain User-Defined Data Types (UDT) available in JDBC API. The UDTs available in the java.sql package are listed in Table 24.4:

Table 24.4: Interfaces for UDT in the java.sql package

Interface	Description
java.sql.SQLData	Provides a mapping between the SQL UDTs and a specific class in Java.
java.sql.SQLInput	Provides methods to read the UDT attributes from a specific input stream. The input stream contains a stream of values depicting the instance of the SQL structured or SQL distinct type.
java.sql.SQLOutput	Writes the attributes of the output stream back to the database.

JDBC API also provides some default data types that are associated with a database. The default types include the DISTINCT and DATALINK types. The DISTINCT data type maps to the base type to which the base type value is mapped. For instance, a DISTINCT value based on an SQL NUMERIC type can be mapped to a java.math.BigDecimal type. A DATALINK type always represents a java.net.URL object of the URL class defined in the java.net package.

Database Metadata

The metadata interface is used to retrieve information about the database used in an application. JDBC API provides certain interfaces to access the information about the database used in the application. These metadata interfaces are described in Table 24.5:

Table 24.5: Interfaces of database metadata

Interface	Description
`java.sql.DatabaseMetaData`	This interface provides the comprehensive information about the database.
`java.sql.ParameterMetaData`	The object of this interface is used to retrieve information about the types and properties for each parameter provided in a `PreparedStatement` object .
`java.sql.ResultSetMetaData`	The object of this interface is used to fetch the information about the types and properties of the columns specified in a `ResultSet` object .

Exceptions and Warnings

JDBC API provides classes and interfaces to handle the unwanted exceptions raised in an application. The API also provides classes to handle warnings related to an application.

Table 24.6 describes the classes for exception handling:

Table 24.6: Classes for exception handling

Classes	Description
`java.sql.BatchUpdateException`	Updates batches.
`java.sql.DataTruncation`	Identifies data truncation errors. Note that data types do not always match between Java and SQL.
`java.sql.SQLException`	Represents all JDBC-related exception conditions. This exception also embeds all driver and database-level exceptions and error codes.
`java.sql.SQLWarning`	Represents database access warnings. Instead of catching the SQLWarning exception, you can use the appropriate methods on the `java.sql.Connection`, `java.sql.Statement`, and `java.sql.ResultSet` objects to access the warnings.

Let's now briefly discuss the JDBC extension APIs (`javax.sql`) available in JDBC API.

*The **javax.sql** Package*

The `javax.sql` package is also called the JDBC extension API, and provides classes and interfaces to access server-side data sources and process Java programs. The JDBC extension package supplements the `java.sql` package and provides the following supports:

- DataSource
- Connection and statement pooling
- Distributed transaction
- Rowsets

DataSource

The `javax.sql.DataSource` interface represents the data sources related to the Java application.

Table 24.7 describes the interfaces of the `DataSource` provided by the `javax.sql` package:

Table 24.7: Interfaces for DataSource

Interface	Description
`javax.sql.DataSource`	Represents the `DataSource` interface used in an application.
`javax.sql.CommonDataSource`	Provides the methods that are common between the DataSource, `XADataSource` and `ConnectionPoolDataSource` interfaces.

Connection and Statement Pooling

The connections made by using the `DataSource` objects are implemented on the middle-tier connection pool. As a result, the functionality to create new database connections is improved. The classes and interfaces available for connection pooling in the `javax.sql` package are listed in Table 24.8:

Table 24.8: Classes and interfaces for connection pooling

Class/Interface	Description
`javax.sql.ConnectionPoolDataSource interface`	Provides a factory for the `PooledConnection` objects.
`javax.sql.PooledConnection interface`	Provides an object to manage connection pools.
`javax.sql.ConnectionEvent class`	Provides an Event object, which offers information about the occurrence of an event.
`javax.sql.ConnectionEventListener inteface`	Provides the objects used to register the events generated by the `PooledConnection` object.
`javax.sql.StatementEvent class`	Represents the `StatementEvent` interface associated with the events that occur in the `PooledConnection` interface. The `StatementEvent` interface is then sent to the `StatementEventListeners` instance, which is registered with the instance of the `PooledConnection` interface.
`javax.sql.StatementEventListener interface`	Provides an object that registers the event with an instance of the `PooledConnection` interface.

Distributed Transaction

The distributed transaction mechanism allows an application to use the data sources on multiple servers in a single transaction. JDBC API provides certain classes and interfaces to handle distributed transactions over the middle-tier architecture, as listed in Table 24.9:

Table 24.9: Interfaces for distributed transaction

Interface	Description
`javax.sql.XAConnection`	Provides the object that supports distributed transaction over middle-tier architecture.
`javax.sql.XADataSource`	Provides a factory for the `XAConnection` objects.

Rowsets Object

A `RowSet` object is used to retrieve data in a network. In addition, the `RowSet` object is able to transmit data over a network. JDBC API provides the `RowSet` interface, with its numerous classes and interfaces, to work with tabular data, as described in Table 24.10:

Table 24.10: Classes and interfaces for RowSet

Class/Interface	Description
`javax.sql.RowSetListener interface`	Receives notification from the `RowSet` object on the occurrence of an event.
`javax.sql.RowSetEvent class`	Provides the Event object, which is generated on the occurrence of an event on the `RowSet` object.
`javax.sql.RowSetMetaData interface`	Contains information about the columns specified in a RowSet object.
`javax.sql.RowSetReader interface`	Populates the disconnected RowSet object with rows of data.

Table 24.10: Classes and interfaces for RowSet	
Class/Interface	**Description**
`javax.sql.RowSetWriter interface`	Implements the `RowSetWriter object, which is also` called `RowSet` writer.
`javax.sql.RowSet interface`	Adds support to the JDBC API for the JavaBeans™ component model.

JDBC versus ODBC and Other APIs

JDBC is a Java API that is used to connect and then execute a query to a database. Earlier, ODBC was used to create and execute the query with the use of ODBC driver written in C language, which is an unsecured and platform-dependent language. Therefore, Oracle defined its own API (JDBC API) that also uses its driver (JDBC driver) written in Java language.

From a programming point of view, the JDBC API is a set of Java classes and interfaces that allow embedding of database calls in server applications. In particular, the JDBC Specification is a set of interfaces that every JDBC driver vendor must execute. A driver processes the JDBC statements in your application and directs the SQL arguments they have to your database engines. Various JDBC drivers are supported by the Sun Java System Application Server. These JDBC drivers support different types of Enterprise Information System (EIS) databases. Figure 24.2 explains how application components use the JDBC API to interact with databases:

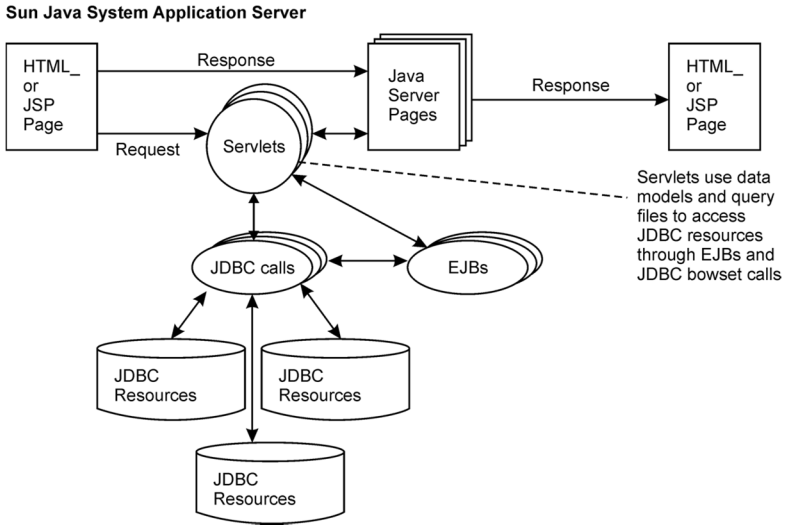


Figure 24.2: Application components using the JDBC API to interact with databases

Relation between JDBC and ODBC

Let's have another look at how JDBC is related to ODBC. The fundamental principle behind the designing of JDBC was to make it practical for building JDBC drivers that are based on APIs of other database. All APIs are based on the same standard, SQL X/Open CLI, fundamentally. So there exists a very close mapping between the JDBC architecture and API and the ODBC counterparts; however, JDBC is easier to use. Table 24.11 details the components for JDBC and ODBC:

Table 24.11: Showing the components for JDBC and ODBC with description	
Component	**Description**
`DriverManager`	It loads database drivers and manages the connection between the application and the driver.

Table 24.11: Showing the components for JDBC and ODBC with description	
Component	**Description**
`Driver`	It translates API calls into operation for a specific data source.
`Connection`	It is a session between an application and a database.
`Statement`	It is a SQL statement to perform a query or update operation.
`MetaData`	It gives information about returned data, the database, and the driver.
`ResultSet`	It is the logical set of columns and rows of data returned by executing a statement.

Various Kinds of JDBC Drivers

The different types of drivers available in JDBC are listed in Table 24.12:

Table 24.12: Types of JDBC drivers	
JDBC Driver Types	**Description**
`Type-1 Driver`	Refers to the Bridge Driver (JDBC-ODBC bridge). The JDBC-ODBC bridge driver has been removed from JDBC 4.2 API.
`Type-2 Driver`	Refers to a partly Java and partly native code driver (Native-API Partly Java driver).
`Type-3 Driver`	Refers to a pure Java driver that uses a middleware driver to connect to a database (Pure Java Driver for Database Middleware).
`Type-4 Driver`	Refers to a Pure Java driver (Pure), which is directly connected to a database.

Now let's discuss each of these drivers in detail.

Describing the Type-1 Driver

The Type-1 driver functions as a channel between the mechanism of JDBC and other database connectivity, such as ODBC. An example of this type of driver is the Sun JDBC-ODBC bridge driver, which provides access to the database through the ODBC drivers. This driver also helps the Java programmers to use JDBC and develop Java applications to communicate with existing data sources. Java 2 SDK includes Type-1 driver within the `sun.jdbc.odbc` package. This driver transforms JDBC calls into ODBC calls and navigates the request to the ODBC driver. The architecture of the Type-1 driver is shown in Figure 24.3:

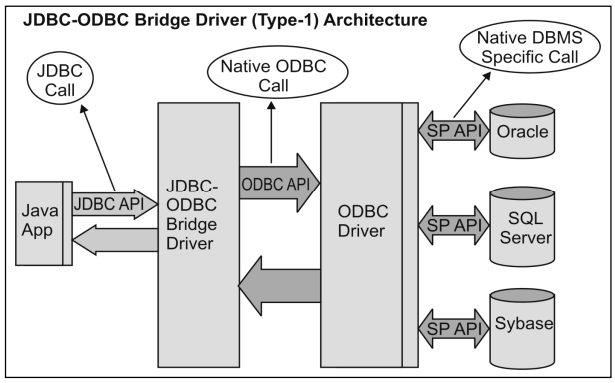


Figure 24.3: Displaying the architecture of the JDBC Type-1 driver

Figure 24.3 shows the architecture of the system that uses the JDBC-ODBC bridge driver to communicate with the respective database. In Figure 24.3, SP API refers to the APIs used to make a Native DBMS specific call.

Figure 24.3 shows the following steps that are involved in establishing connection between a Java application and data source through the Type-1 driver:

1. The Java application makes the JDBC call to the JDBC-ODBC bridge driver to access a data source.
2. The JDBC-ODBC bridge driver resolves the JDBC call and makes an equivalent ODBC call to the ODBC driver.
3. The ODBC driver completes the request and sends responses to the JDBC-ODBC bridge driver.
4. The JDBC-ODBC bridge driver converts the response into JDBC standards and displays the result to the requesting Java application.

The Type-1 driver is generally used in the development and testing phases of Java applications.

Advantages of the Type-1 Driver

Some advantages of the Type-1 driver are as follows:

- Represents single driver implementation to interact with different data stores
- Allows us to communicate with all the databases supported by the ODBC driver
- Represents a vendor independent driver

Disadvantages of the Type-1 Driver

Some disadvantages of the Type-1 driver are as follows:

- Decreases the execution speed due to a large number of translations
- Depends on the ODBC driver; and therefore, Java applications also become indirectly dependent on ODBC drivers
- Requires the ODBC binary code or ODBC client library that must be installed on every client
- Uses Java Native Interface (JNI) to make ODBC calls

The preceding disadvantages make the Type-1 driver unsuitable for production environment and should be used only in case when no other driver is available. The Type-1 driver is not recommended either when Java applications are required with auto-installation applications, such as applets.

Describing the Type-2 Driver (Java to Native API)

Type-2 driver helps in converting the JDBC call into the database-vendor-specific native call. In other words, this type of driver makes the Java Native Interface (JNI) calls on database-specific native client API. These database-specific native client APIs are usually written in C and C++.

The Type-2 driver follows a 2-tier architecture model, as shown in Figure 24.4:

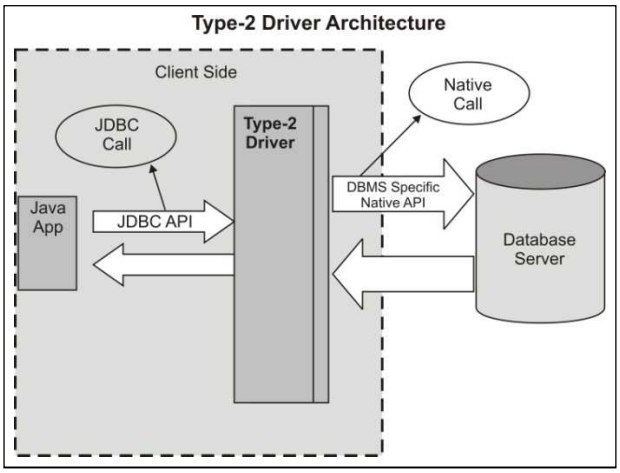


Figure 24.4: Displaying the architecture of the JDBC Type-2 driver

As shown in Figure 24.4, the Java application that needs to communicate with the database is programmed using JDBC API. In the client machine, the JDBC calls (programs created by using JDBC API) are then transformed into database-specific native calls and the request is then sent to the database-specific native libraries. These libraries present in the client are intelligent enough to send the request to the database server by using native protocol.

This type of driver is implemented for a specific database and usually delivered by a DBMS vendor. However, it is not mandatory that Type-2 drivers have to be implemented by DBMS vendors only. An example of Type-2 driver is the Weblogic driver implemented by BEA Weblogic. Type-2 drivers can be used with server-side applications. It is not recommended to use Type-2 drivers with client-side applications, since the database specific native libraries should be installed on the client machines.

Advantages of the Type-2 Driver

Some advantages of the Type-2 driver are as follows:

- Helps to access the data faster as compared to other types of drivers
- Contains additional features provided by the specific database vendor, which are also supported by the JDBC specification

Disadvantages of the Type-2 Driver

Some disadvantages of the Type-2 driver are as follows:

- Requires native libraries to be installed on client machines, since the conversion from JDBC calls to database specific native calls is done on client machines
- Executes the database-specific native functions on the client JVM, implying that any bug in the Type-2 driver might crash the JVM
- Increases the cost of the application in case it is run on different platforms

Examples of the Type-2 Driver

Some examples of the Type-2 driver are as follows:

- **OCI (Oracle Call Interface) Driver**—Communicates with the `Oracle` database server. This driver converts JDBC calls into `Oracle` native library calls.
- **Weblogic OCI Driver for Oracle**—Makes JNI calls to Weblogic library functions. The Weblogic OCI driver for Oracle is similar to the Oracle OCI driver.
- **Type-2 Driver for Sybase**—Converts JDBC calls into Sybase dblib or ctlib calls, which are native libraries to connect to Sybase.

Describing the Type-3 Driver (Java to Network Protocol/All Java Driver)

The Type-3 driver is used for translating the JDBC calls into calls that are database server independent and middleware server specific. The translated JDBC calls are then translated into database-server-specific calls with the help of the middleware server.

The Type-3 drivers follow the 3-tier architecture model, as shown in Figure 24.5:

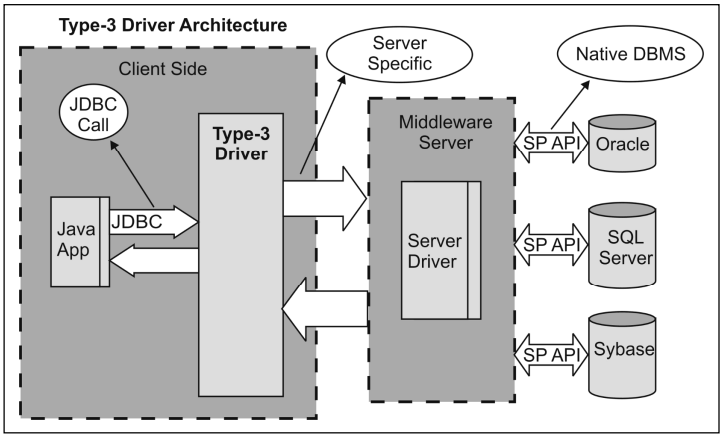


Figure 24.5: Displaying the architecture of the JDBC Type-3 driver

As shown in Figure 24.5, a JDBC Type-3 driver listens for JDBC calls from the Java application and translates them into middleware server specific calls. After that, the driver communicates with the middleware server over a socket. The middleware server converts these calls into database specific calls. These types of drivers are also called *net-protocol fully Java technology-enabled* or *net-protocol* drivers.

The middleware server can be added in an application with some additional functionality, such as pool management, performance improvement, and connection availability. These functionalities make the Type-3 driver architecture more useful in enterprise applications. Since Type-3 driver is auto-downloadable, it is recommended to be used with applets.

Advantages of the Type-3 Drivers

Some advantages of the Type-3 driver are as follows:

- Serves as all Java driver and is auto-downloadable.
- Does not require any native library to be installed on the client machine.
- Ensures database independency because a single driver provides accessibility to different types of databases.
- Does not provide the database details, such as username, password, and database server location, to the client. These details are automatically configured in the middleware server.
- Provides the facility to switch over from one database to another without changing the client-side driver classes. Switching of databases can be implemented by changing the configurations of the middleware server.

Disadvantage of the Type-3 Driver

The main disadvantage of the Type-3 driver is that it performs the tasks slowly due to the increased number of network calls as compared to Type–2 drivers. In addition, the Type-3 driver is costlier as compared to other drivers.

Examples of the Type-3 Drivers

Some examples of the Type-3 driver are as follows:

- **IDS Driver—** Listens for JDBC calls and converts them into IDS Server specific network calls. The Type-3 driver communicates over a socket to IDS Server, which acts as a middleware server.
- **Weblogic RMI Driver—** Listens for JDBC calls and sends the requests from the client to the middleware server by using the RMI protocol. The middleware server uses a suitable JDBC driver to communicate with a database.

Describing the Type-4 Driver (Java to Database Protocol)

The Type-4 driver is a pure Java driver, which implements the database protocol to interact directly with a database. This driver does not need any native database library for retrieving the records stored in the database. In addition, the Type-4 driver translates JDBC calls into database specific network calls.

The Type-4 drivers follow the 2-tier architecture model, as shown in Figure 24.6:

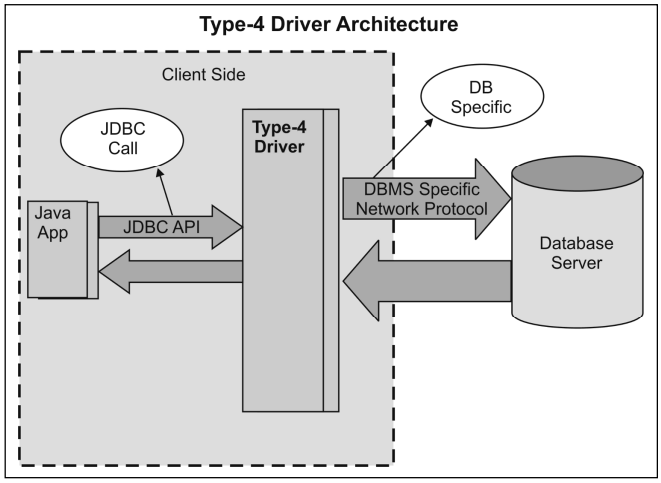


Figure 24.6: Displaying the architecture of the JDBC Type-4 driver

As shown in Figure 24.6, the Type-4 driver prepares a DBMS-specific network message and then communicates with database server over a socket. This type of driver is lightweight and generally known as a thin driver. The Type-4 driver uses database-specific proprietary protocols for communication. Generally, this type of driver is implemented by DBMS vendors since the protocols used are proprietary.

You can use the Type-4 driver when you want an auto-downloadable option for the client-side applications. In addition, it can be used with server-side applications.

Advantages of the Type-4 Driver

Some advantages of the Type-4 driver are as follows:

- Serves as a pure Java driver and is auto-downloadable
- Does not require-any native library to be installed on the client machine
- Uses database-server-specific protocol
- Does not require a middleware server

Disadvantage of the Type-4 Driver

The main disadvantage of the Type-4 driver is that it uses database-specific proprietary protocol and is DBMS vendor dependent.

Examples of the Type-4 Driver

Some examples of the Type-4 driver are:

- Thin Driver for `Oracle` from `Oracle` Corporation
- Weblogic and Mssqlserver4 for MS SQL Server from BEA systems

The JavaSoft Framework

"Uh-Oh," the Novice Programmer says, "I find it very peculiar to work with Java. What about the different product components of JDK without which our users will face difficulties? I want to know more about it." "Ok," you say. "Let me explain three such kinds of Java Development Kits."

The three JDBC product components provided by JavaSoft as a part of the Java Development Kit (JDK) are as follows:

- The JDBC Driver Manager
- The JDBC Driver Test Suite
- The JDBC-ODBC Bridge

The JDBC driver manager is the backbone of the JDBC architecture. It is very small and simple. The basic function of this is to connect Java applications to the correct JDBC driver and subsequently get out of the way.

The second one, i.e., JDBC driver test suite, enables the JDBC drivers to run your program. The designation "JDBC Complaint™" should only be used by the drivers that have passed the JDBC driver test suite.

The third one, i.e., the JDBC-ODBC Bridge, allows ODBC drivers to be used as JDBC drivers. The javasoft framework is shown in Figure 24.7:

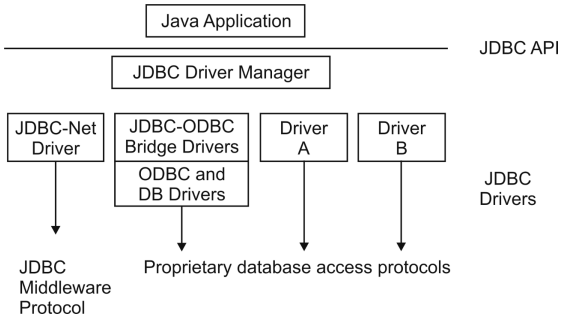


Figure 24.7: Showing the Javasoft framework

Two-Tier and Three-Tier Models

JDBC API supports both two-tier and three-tier architectures to access data from the database.

In two-tier model, Java application communicates directly to the data source. This needs a JDBC driver that can communicate with the intended data source being accessed. A user's commands are sent to the database or other data source, and the output is sent back to the user. The data source may be located on another remote machine to which the user is interacting in the network. It acts as client/server model in which the user's machine acts as the client and the remote machine that hosts the data source acts as the server. The network can be an intranet, which, for example, connects students of different branches of same college. A type-2 driver following a two-tier architecture is shown in Figure 24.8:

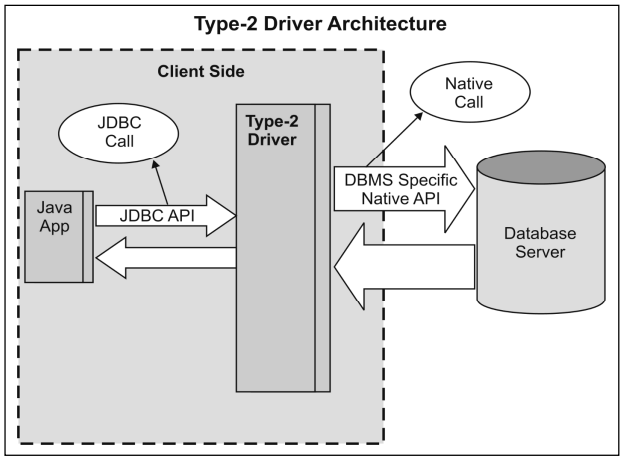


Figure 24.8: Two-tier architecture

In the three-tier model, a middle tier exists, which acts as a bridge between the Java application and the data source. Then, when responding back, the data source sends the result to the middle-tier, which then transmits it to the user. Management Information System directories find the three-tier model useful because the middle tier allows control over access. Another benefit is that it simplifies the deployment of applications. Also, three-tier architecture has a better performance than two-tier architecture. Earlier, languages such as C and C++ are used for creating the middle tier which provides fast performance in accessing data from database. Using optimizing compilers for translating Java bytecode into machine-specific code and technologies such as Enterprise Java Beans, the Java platform is used as the standard platform for middle-tier development. It provides Java with robustness, multithreading, and security features. The three-tier architecture is shown in Figure 24.9:

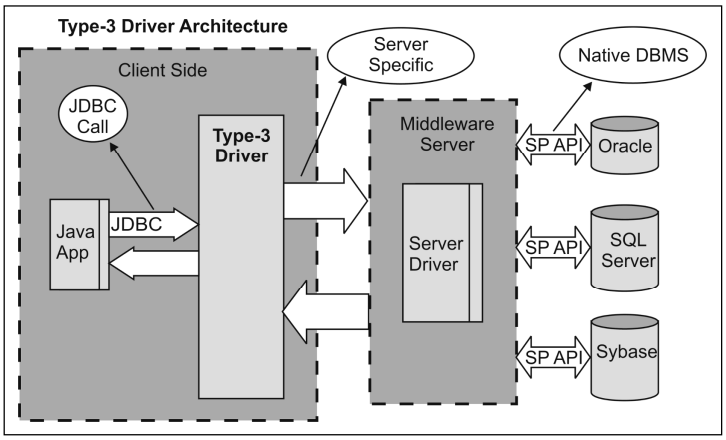


Figure 24.9: Three-tier architecture

The Structured Query Language (SQL)

Java uses the concept of annotation while using SQL, which ultimately results in fewer lines of code. The annotations are used along with the queries. The query results can be bound to the Java classes to speed up the processing of the query output. The JDBC specification provides two main annotations:

- **The SELECT annotation**—Retrieves query-specific data from a database. You can use the SELECT annotation in a SELECT query within a Java class. The attributes of the SELECT annotation are described in Table 24.13:

Table 24.13: Attributes of the SELECT Annotation

Name	Type	Description
`sql`	String	Specifies a simple SQL SELECT query.
`value`	String	Represents the value specified for the SQL attribute.
`tableName`	String	Specifies the name of the table created in a database.
`readOnly, connected, scrollable`	boolean	Indicates whether DataSet is ReadOnly or Updatable. It also indicates whether or not DataSet is connected to a back-end database. In addition, it indicates whether or not it is scrollable when the query is used in a connection.
`allColumnsMapped`	boolean	Indicates whether or not the column names used in the annotations are mapped to the corresponding fields in DataSet.

- **The UPDATE annotation**—Updates the queries used in database tables. The `UPDATE` annotation must include the SQL annotation type to update the fields of a table.

There are certain enhancements in the `SQLException` class, which are as follows:

- **New exception subclasses**—Provide new classes as enhancement to `SQLException`. The new classes that are added to the `SQLException` exception class include SQL non-transient exception and SQL transient exception. The SQL non-transient exception class is called when an already performed JDBC operation fails to run unless the cause of the `SQLException` exception is corrected. On the other hand, the SQL transient exception class is called when a previously failed JDBC operation succeeds after retry.
- **Casual relationships**—Support the mechanism of Java SE chained exception by the `SQLException` class (also called Casual facility). It allows handling multiple SQL exceptions raised in the JDBC operation.
- **Support for the for-each loop**—Implements the chain of exceptions in a chain of groups by the `SQLException` class. The `for-each` loop is used to iterate on these groups.
- **SQL XML support**—Introduces the concept of `XML` support in SQL DataStore. Some additional APIs have been added to JDBC 4.0 to provide this support.

Immediate Solutions

Learning about the Driver Interface

The `Driver` interface is used to create connection objects that provide an entry point for database connectivity. Generally, all drivers provide the `DriverManager` class that implements the `Driver` interface and helps to load the driver in a JDBC application. The drivers are loaded for any given connection request with the help of the `DriverManager` class. After the driver class is loaded, its instance is created and registered with the `DriverManager` class.

Table 24.14 describes all the methods provided in the `Driver` interface:

Table 24.14: Methods of the Driver interface

Method	Description
`boolean acceptsURL(String url)`	Checks whether or not the specified URL format is acceptable by the driver.
`Connection connect(String url, Properties info)`	Establishes the connection with the database by using the given URL.
`int getMajorVersion()`	Accesses the major version number of the driver.
`int getMinorVersion()`	Retrieves the minor version number of the driver.
`DriverPropertyInfo[] getPropertyInfo(String url, Properties info)`	Retrieves the properties of the driver on the basis of the given URL.
`boolean jdbcCompliant()`	Reports whether the current driver is a genuine JDBC Compliant™ driver.
`Logger getParentLogger()`	Returns the parent Logger of all the Loggers used by this driver.

Learning about the DriverManager Class

`DriverManager` is a non-abstract class in JDBC API. It contains only one constructor, which is declared private to imply that this class cannot be inherited or initialized directly. All the methods and properties of this class are declared as static. The `DriverManager` class performs the following main responsibilities:

- Maintains a list of `DriverInfo` objects, where each `DriverInfo` object holds one driver implementation class object and its name
- Prepares a connection using the driver implementation that accepts the given JDBC URL

Table 24.15 describes the methods of the `DriverManager` class:

Table 24.15: Methods of the DriverManager class

Method	Description
`static void deregisterDriver(Driver driver)`	Drops a driver from the list of drivers maintained by the `DriverManager` class
`static Connection getConnection(String url)`	Establishes a connection of a driver with a database. The `DriverManager` class selects a driver from the list of drivers and creates the connection
`static Connection getConnection(String url, Properties info)`	Establishes a connection of a driver with a database on the basis of the URL and info passed as parameters. URL is used to load the selected driver for a database. The info parameter provides information about the string/value tags used in the connection

Table 24.15: Methods of the DriverManager class

Method	Description
`static Connection getConnection(String url, String username, String password)`	Establishes a connection of a driver with a database. The `DriverManager` class selects a driver from the list of drivers and creates the connection. Along with URL, it takes two more parameters, that is, username and password. The username parameter specifies the user for which the connection is being made, and the password parameter represents the password of the user
`static Driver getDriver(String url)`	Locates the requested driver in the `DriverManager` class. The url parameter specifies the URL of the requested driver.
`static Enumeration<Driver>getDrivers()`	Accesses a list of drivers present in a database.
`static int getLoginTimeout()`	Specifies the maximum time a driver needs to wait to log on to a database.
`static PrintStream getLogStream()`	Returns the logging or tracing `PrintStream` object.
`static PrintWriter getLogWriter()`	Returns the log writer.
`static void println(String message)`	Prints a message used in a log stream.
`static void registerDriver(Driver driver)`	Registers a requested driver with the `DriverManager` class.
`static void setLoginTimeout(int seconds)`	Sets the maximum time that a driver needs to wait while attempting to connect to a database.
`static void setLogStream(PrintStream out)`	Sets the logging or tracing `PrintStream` object.
`static void setLogWriter(PrintWriter out)`	Sets the logging or tracing `PrintWriter` object.

The Connection Interface

The `Connection` interface is used for defining an abstraction for accessing the connection established with a database server. JDBC driver provider must implement the `Connection` interface. The `Connection` type of object (an instance of the class that implements the `Connection` interface) represents the session established with the data store. The `Connection` interface provides methods to handle the `Connection` object.

Table 24.16 describes the methods present in the `Connection` interface:

Table 24.16: Methods of the Connection interface

Method	Description
`void clearWarnings()`	Clears all the warnings for a `Connection` object. This method throws the `SQLException` exception when an error occurs.
`void close()`	Closes a connection and releases the connection object associated with the connected database. It also releases the JDBC resources associated with the connection.
`void commit()`	Commits the changes made in the previous commit/rollback and releases any database locks held by the current Connection object .
`Statement createStatement()`	Creates the `Statement` object to send SQL statements to the specified database. This method takes no argument; therefore, it can be executed by using the Statement object.

Table 24.16: Methods of the Connection interface	
Method	**Description**
`Statement createStatement(int resultSetType, int resultSetConcurrency)`	Creates a `Statement` object, which is used to load the SQL statements to the specified database. The `ResultSet` object generated by this `Statement` object is of the mentioned type and concurrency.
`Statement createStatement (int resultSetType, int resultSetConcurrency, int resultSetHoldability)`	Creates an object with the mentioned type, concurrency, and holdability.
`boolean getAutoCommit()`	Retrieves the auto-commit mode for the current `Connection` object.
`String getCatalog()`	Gets the name of the current catalog used in the current `Connection` object.
`int getHoldability()`	Gets the current holdability of the `ResultSet` object created by using a `Connection` object.
`DatabaseMetaData getMetaData()`	Gets the `DatabaseMetadata` object containing the metadata information. You should ensure that the database must be connected with a connection object.
`int getTransactionIsolation()`	Provides the transaction isolation level of the `Connection` object related to a database.
`Map getTypeMap()`	Gets a map object related to a `Connection` object.
`SQLwarning getWarnings()`	Retrieves any warning associated with a `Connection` object.
`boolean isClosed()`	Specifies whether or not a database `Connection` object is closed.
`boolean isReadOnly()`	Specifies whether or not a `Connection` object is read only.
`String nativeSQL(String sql)`	Allows you to convert the SQL statements passed to the `Connection` object into the systems native SQL grammar.
`CallableStatement prepareCall(String sql)`	Creates a CallableStatement object to call database-stored procedures.
`CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency)`	Creates a `CallableStatement` object that generates the ResultSet object of the specified type and concurrency.
`CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)`	Creates a `CallableStatement` object that generates the ResultSet object of the specified type, concurrency, and holdability.
`PreparedStatement prepareStatement(String sql)`	Creates a `PreparedStatement` object to send the SQL statements over a connection.
`PreparedStatement prepareStatement(String sql, int autoGeneratedKeys)`	Creates a `PreparedStatement` object that retrieves auto-generated keys.
`PreparedStatement prepareStatement(String sql, int[] columnIndexes)`	Creates a `PreparedStatement` object that retrieves auto-generated keys by using a given array.

Table 24.16: Methods of the Connection interface

Method	Description
`PreparedStatement prepareStatement(String sql, int resultSetType, int resultSetConcurrency)`	Generates a `PreparedStatement` object that generates the `ResultSet` object with the given type and concurrency.
`PreparedStatement prepareStatement(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)`	Generates a `PreparedStatement` object that generates the `ResultSet` object with the given type, concurrency, and holdability.
`PreparedStatement prepareStatement(String sql, String[] columnNames)`	Creates a `PreparedStatement` object that retrieves the auto-generated keys. The columnNames parameter of `PreparedStatement` is an array containing the names of the columns that contain the auto-generated keys in the target table.
`void releaseSavepoint(Savepoint savepoint)`	Releases the savepoint associated with the connection object of the current transaction.
`void rollback ()`	Rolls back all the transactions and releases any database locks that are currently done by the connection object.
`void rollback(Savepoint savepoint)`	Rollbacks all the changes made by the connection object after a savepoint object is created.
`void setAutoCommit(boolean autoCommit)`	Sets the current transaction to the connections auto-commit mode.
`void setCatalog(String catalog)`	Sets the given catalog name for the current `Connection` object's database.
`void setHoldability(int holdability)`	Changes the holdability of the current `Connection` object.
`void setReadOnly(boolean readOnly)`	Sets the connection to the read-only mode to optimize the specified database.
`Savepoint setSavepoint()`	Creates an unnamed savepoint in the current transaction and returns the savepoint associated with the previous transactions.
`Savepoint setSavepoint(String name)`	Creates a savepoint with the name specified in the current transaction. It returns the new savepoint object
`void setTransactionIsolation(int level)`	Checks the transaction isolation level of the specified `Connection` object.
`void setTypeMap(Map<String, Class<?>> map)`	Installs the `TypeMap` object as the current type map for the current connection.
`Array createArrayOf(String typeName, Object[] elements)`	Refers to a `Factory` method used to create `Array` objects.
`Blob createBlob()`	Creates an object implementing the `Blob` interface.
`Clob createClob()`	Creates an object implementing the `Clob` interface.
`NClob createNClob()`	Creates an object implementing the `NClob` interface.
`SQLXML createSQLXML()`	Creates an object implementing the `SQLXML` interface.
`Struct createStruct(String typeName, Object[] attributes)`	Refers to the factory method that creates `Struct` objects.

The `Connection` interface also provides certain fields that can be used to handle connection transactions.

Table 24.17 describes the fields available in the `Connection` interface:

Table 24.17: Fields of the Connection interface

Field	Description
static int TRANSACTION_NONE	Refers to a constant representing that transactions are not supported.
static int TRANSACTION_READ_COMMITTED	Prevents a transaction from reading a row with uncommitted changes. It is only used to read non-repeatable rows in a table.
static final int TRANSACTION_READ_UNCOMMITTED	Indicates that non-repeatable and phantom reads are allowed in a transaction. It allows a row to be changed during a transaction. The changed row can be read by other transactions before the changes in the row are committed.
static int TRANSACTION_REPEATABLE_READ	Prevents non-repeatable reads and simultaneous transactions in a single row.
static int TRANSACTION_SERIALIZABLE	Prevents reading non-repeatable rows in a table.

Let's now learn about the `Statement` interface.

The Statement Interface

The `Statement` interface is used for defining a standard abstraction to run the SQL statements, which are requested by a user and provides the results with the help of the `ResultSet` object. The `Statement` object contains a single `ResultSet` object at a time. It might be possible that the reading of data performed by using one `ResultSet` object is incorporated with the reading done by the other. In such a case, each `ResultSet` object must be generated by different `Statement` objects. The `execute()` method of all the statements implicitly closes the current `ResultSet` object (if it is open) of a statement. The `Statement` interface provides specific methods to execute and retrieve the results from a database. The methods provided by the `PreparedStatement` interface are used for dealing with the `IN` parameters, whereas the methods provided by the `CallableStatement` interface are used for dealing with the `IN` and `OUT` parameters.

The `Statement` interface also provides certain methods that are used with a database. These methods are described in Table 24.18:

Table 24.18: Methods of the Statement interface

Method	Description
`void addBatch(String sql)`	Adds the SQL commands to the existing list of commands for the Statement object. These commands are executed in a batch by calling the executeBatch() method.
`void abort(Executor executor)`	Terminates an open connection with the database.
`void cancel()`	Cancels the statement if the data sources do not support the statement.
`void clearBatch()`	Clears all the commands listed in the batch of the Statement object.
`void clearWarnings()`	Clears the warnings that are generated on the Statement object. You should note that after the execution of the clearWarnings() method, the `getWarnings()` method returns null, provided a new warning is not generated for this Statement object.
`void close()`	Closes the Statement object. Therefore, it releases its control from the database and connection.
`boolean execute(String sql)`	Executes the SQL commands that may return multiple result sets along with one or more update counts.
`boolean execute(String sql, int autoGeneratedKeys)`	Executes the SQL commands that may return multiple result sets along with one or more update counts. It also indicates whether a driver or the auto-generated keys are available for the retrieval.

Table 24.18: Methods of the Statement interface

Method	Description
`boolean execute(String sql, int[] columnIndexes)`	Executes the SQL commands that may return multiple result sets along with one or more update counts. It also indicates the driver about the availability of the auto-generated keys in an array. The array contains the list of the indexes and the tables containing the auto-generated keys.
`boolean execute(String sql, String[] columnNames)`	Executes the SQL commands that may return multiple result sets along with one or more update counts. It also indicates the driver about the availability of the auto-generated keys in an array. The array contains the name of the columns in the target table that contains the auto-generated keys.
`int[] executeBatch()`	Executes the SQL commands in a batch. The method returns the update count as an integer greater or equal to 0 after the successful execution of the batch statements. The integer array is used to represent the array of the SQL commands listed in the batch.
`ResultSet executeQuery(String sql)`	Executes a SQL command and returns a single ResultSet
`int executeUpdate(String sql)`	Executes the SQL Data Definition Language (DDL) statements, such as `INSERT`, `UPDATE`, and `DELETE`.
`int executeUpdate(String sql, int autoGeneratedKeys)`	Executes the SQL statements and notifies the driver about the availability of the auto-generated keys. The auto-generated keys are helpful to retrieve data from the database
`int executeUpdate(String sql, int[] columnIndexes)`	Executes the SQL statements on the basis of the SQL query and column index passed as an argument. This method also notifies the driver about the availability of the auto-generated keys. The auto-generated keys are helpful to retrieve data from the database. The array index of the auto-generated keys indicates the indexes and tables that contain the auto-generated keys.
`int executeUpdate(String sql, String[] columnNames)`	Executes the SQL statements and notifies the driver about the availability of the auto-generated keys. These keys are responsible for data retrieval from the database. The array index of the auto-generated keys indicates the columns of the target table that contains the auto-generated keys
`Connection getConnection()`	Retrieves an object of Connection type, which is used to maintain the connection of a Java application with a database.
`int getFetchDirection()`	Retrieves the direction of the rows from the database tables that are generated from the ResultSet object.
`int getFetchSize()`	Gets the number of rows of default fetch size from the current ResultSet object.
`ResultSet getGeneratedKeys()`	Gets the auto-generated keys created by executing the Statement object.
`int getMaxFieldSize()`	Gets the maximum number of bytes that can be returned for the column values.
`int getMaxRows()`	Provides the maximum number of rows in a ResultSet object produced by the Statement object.
`boolean getMoreResults()`	Navigates to the next result in the ResultSet object. It is also used to close the currently opened result set.
`int getMoreResults(int current)`	Navigates to the next result in the object of the statement. It deals with the ResultSet object by using the instructions specified in the given flag.
`int getQueryTimeout()`	Provides the number of seconds the driver has to wait to execute the statements.

Table 24.18: Methods of the Statement interface	
Method	**Description**
`ResultSet getResultSet()`	Gets the current ResultSet object generated by the Statement object.
`int getResultSetConcurrency()`	Gets the concurrency of the ResultSet object generated by the Statement object.
`int getResultSetHoldability()`	Gets the holdability of the ResultSet object generated by the Statement object
`int getResultSetType()`	Retrieves the result set type for the ResultSet object
`int getUpdateCount()`	Retrieves the current result set as an update count. The value returned by this method is either a positive or negative value, indicating the number of records that have been updated in a result set.
`SQLWarning getWarnings()`	Gets the warnings generated on the Statement object
`void setCursorName(String name)`	Sets the cursor name to the given string. The cursor name is used by the Statement objects to execute this method.
`void setEscapeProcessing(boolean enable)`	Sets the escape processing on or off.
`void setFetchDirection(int direction)`	Sets the direction for the driver to process the rows in the ResultSet object.
`void setFetchSize(int rows)`	Sets the number of rows that should be fetched from the database.
`void setMaxFieldSize(int max)`	Sets the maximum number of bytes for the ResultSet object to store binary values.
`void setMaxRows(int max)`	Sets the maximum number of rows that a ResultSet object can contain.
`void setQueryTimeout(int seconds)`	Sets the number of seconds a driver needs to wait for executing the Statement object.

The `Statement` interface also comprises few fields. Table 24.19 describes the fields available in the `Statement` interface:

Table 24.19: Fields of the Statement interface	
Field	**Description**
`static int CLOSE_ALL_RESULTS`	Closes all the open `ResultSet` objects. All the ResultSet objects should be closed before calling the `getMoreResults()` method.
`static int CLOSE_CURRENT_RESULT`	Indicates that the current `ResultSet` connected with the specified database must be closed before calling the `getMoreResults()` method.
`static int EXECUTE_FAILED`	Indicates the occurrence of errors while executing a batch statement.
`static int KEEP_CURRENT_RESULT`	Indicates that the current `Resultset` should not be closed before calling the `getMoreResults()` method.
`static int NO_GENERATED_KEYS`	Indicates that the generated keys should not be made available for retrieval.
`static int RETURN_GENERATED_KEYS`	Indicates that the generated keys should be made available for retrieval.
`static int SUCCESS_NO_INFO`	Indicates that a batch statement has been executed successfully.

Table 24.18 shows all the required fields in the `Statement` interface. These are used by a database to communicate with an application.

The `Statement` object is created after the connection to the specified database is made. You can create this object with the help of the `createStatement()` method provided by the `Connection` interface, as shown in the following code snippet:

```
Connection con = DriverManager.getConnection (url, "username", "password");
Statement stmt = con.createStatement();
```

The PreparedStatement Interface

The `PreparedStatement` interface is used for representing a precompiled query that you can run multiple times. Let's now first understand the difference between the execution process of a `Statement` object and the `PreparedStatement` object to execute a JDBC query.

Next, you learn about the `setXXX()` methods as well as the advantages and disadvantages of the `PreparedStatement` interface. You also learn how to implement the `PreparedStatement` interface to execute the SQL query.

Comparing the Execution Control of the Statement and PreparedStatement

When a `Statement` object is used to execute a query (that is, calling any one of the `execute` methods), the query is processed as follows:

1. The `executeXXX()` method is invoked on the `Statement` object by passing the SQL statement as parameter.
2. The `Statement` object submits the SQL statement to the database.
3. The database compiles the given SQL statement.
4. An execution plan is prepared by the database to execute the SQL statements.
5. The execution plan for the compiled SQL statement is then executed. Now, if the SQL statement is a data retrieval statement, such as the SELECT statement, the database caches the results of the SQL statement in the buffer.
6. The results are sent to the `Statement` object.
7. Finally, the response is sent to the Java application in the form of `ResultSet`.

Figure 24.10 displays the entire execution flow of the Statement object:

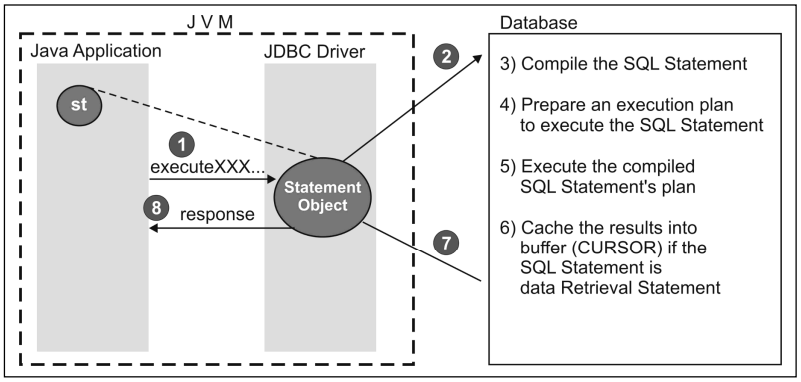


Figure 24.10: Displaying the process flow of the Statement object

In Figure 24.10, the `st` element represents the `Statement` object reference. Compilation of a query includes syntax checking, name validation, and pseudocode generation. After a query is validated, the query optimizer prepares for the execution of the query and then returns what it considers to be the best alternative. The SQL statement needs to be executed each time it is requested.

It is not necessary to compile the SQL statement and prepare execution plan to execute a statement multiple times. DBMSs are designed to store the execution plans and execute them multiple times, if required. Consequently, the processing time of the DBMSs is optimized. These stored execution plans of the SQL

statements are known as pre-compiled SQL statements. DBMS intelligently maintains the compiled queries and provides a unique identity for the prepared execution plan, which the client uses to execute the same query next time. JDBC specifications support the use of this feature provided by DBMS. The `PreparedStatement` interface is designed specifically to support this feature.

The execution of pre-compiled PreparedStatement is much faster in comparison to the Statement objects included in an application. The `PreparedStatement` interface inherits all the properties of the `Statement` interface. The execute methods do not take any parameter while using the `PreparedStatement` object.

You should keep in mind the following points while using the PreparedStatement interface:

- A `PreparedStatement` object must be associated with one connection
- A `PreparedStatement` object specifies the running plan of a query, which is then passed as argument during the creation of the `PreparedStatement` object
- After the connection on which the `PreparedStatement` object was created is closed, `PreparedStatement` is implicitly closed
- When a `PreparedStatement` object is used to execute a query (that is, calling any one of the execute methods), the query is processed as follows:
  - The `prepareStatement()` method of the connection object is used to get the object of the `PreparedStatement` interface.
  - The connection object submits the given SQL statement to the database.
  - The database compiles the given SQL statement.
  - An execution plan is prepared by the database to execute the SQL statements.
  - The database stores the execution plan with a unique ID and returns the identity to the Connection object.
  - The `Connection` object prepares a `PreparedStatement` object, initializes it with the execution plan identity, and returns the reference of the `PreparedStatement` object to the Java application.
  - The setXXX() methods of the `PreparedStatement` object are used to set the parameters of the SQL statement it is representing .
  - The executeXXX() method of the `PreparedStatement` object is invoked to execute the SQL statement with the parameters set to the `PreparedStatement` object
  - The `PreparedStatement` object delegates the request sent by a client to the database.
  - The database locates and executes the execution plan with the given parameters.
  - Finally, the result of the SQL statements is sent to the Java application in the form of `ResultSet`.

Figure 24.11 explains the flow of execution when `PreparedStatement` is used to execute SQL statements:

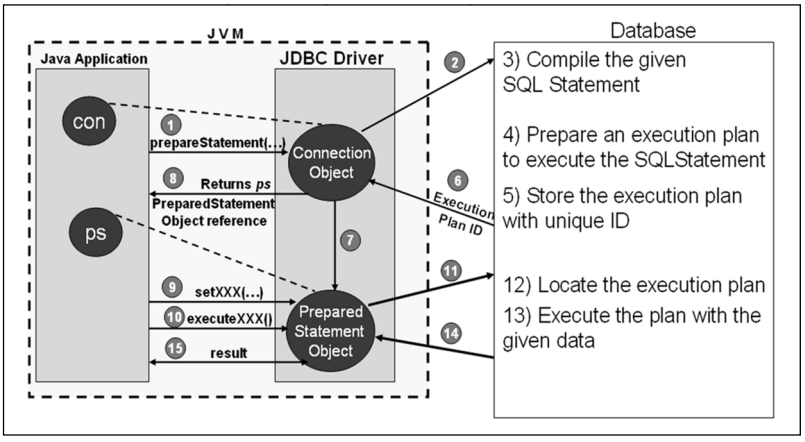


Figure 24.11: Showing steps involved in using the PreparedStatement Object

In Figure 24.11, the `con` and `ps` elements represent the references of the `Connection` and `PreparedStatement` objects, respectively.

Describing the setXXX() Methods of the PreparedStatement Interface

You need to set the value of each placeholder ('?') parameter that is used inside the query string before executing a `PreparedStatement` object. The values for these placeholder parameters are provided at runtime to the SQL queries used within the `PreparedStatement` object. The values of this parameter can be set by using the `setXXX()` methods.

Table 24.20 describes the `setXXX()` methods of the `PreparedStatement` interface:

Table 24.20: Methods available in the PreparedStatement interface

Method	Description
`void setArray(int parameterIndex, Array x)`	Sets the values of parameters to the given array object.
`void setAsciiStream(int parameterIndex, InputStream x, int length)`	Sets the values of the PreparedStatement parameter according to the given input stream specified in the method.
`void setBigDecimal(int parameterIndex, BigDecimal x)`	Sets the values of the parameter by using the values specified in the `java`.math.BigDecimal value
`void setBinaryStream(int parameterIndex, InputStream x, int length)`	Sets the binary values for the parameters used in the PreparedStatement object.
`void setBlob(int parameterIndex, Blob x)`	Sets an integer value to the specified Blob object.
`void setBoolean(int parameterIndex, boolean x)`	Sets the boolean values for the parameters used in the PreparedStatement object.
`void setByte(int parameterIndex, byte x)`	Sets the byte values for the parameters used in the PreparedStatement object.
`void setBytes(int parameterIndex, byte[] x)`	Sets the byte values in an array for the parameters used in the PreparedStatement object.
`void setCharacterStream(int parameterIndex , Reader reader, int length)`	Sets the character values for the PreparedStatement parameters and also specifies the length of the characters.
`void setClob(int parameterIndex , Clob x)`	Sets an integer value to the specified Clob object.
`void setDate(int parameterIndex, Date x)`	Sets the PreparedStatement parameter with a `java.sql.Date` value.
`void setDate(int parameterIndex, Date x, Calendar cal)`	Sets the PreparedStatement parameter with a `java.sql.Date` value and also uses the Calendar object to set the value of the parameter.
`void setDouble(int parameterIndex, double x)`	Sets the value of the parameter to the `Java` double value.
`void setFloat(int parameterIndex, float x)`	Sets the value of the parameter to the `Java` float value.
`void setInt(int parameterIndex, int x)`	Sets the value of the parameter to the `Java` int value.
`void setLong(int parameterIndex, long x)`	Sets the value of the parameter to the `Java` long value.

Table 24.20: Methods available in the PreparedStatement interface	
Method	**Description**
`void setNull(int parameterIndex, int sqlType)`	Sets the NULL values for the parameters of the specified sqlType.
`void setNull(int parameterIndex, int sqlType, String typeName)`	Sets the NULL values for the parameters of the specified sqlType and typeName.
`void setObject(int parameterIndex, Object x)`	Sets the value of the parameter by using the given object value.
`void setObject(int parameterIndex, Object x, int targetSqlType)`	Sets the value of the parameter by using the given object value.
`void setObject(int parameterIndex, Object x, int targetSqlType, int scaleOrLength)`	Sets the value of the parameter by using the given object value
`void setRef(int parameterIndex , Ref x)`	Sets the values of the parameters to the REF (<structured-type>) value.
`void setShort(int parameterIndex, short x)`	Sets the value of the parameter to the `Java` short value.
`void setString(int parameterIndex, String x)`	Sets the value of the parameter to the `Java String` value.
`void setTime(int parameterIndex, Time x)`	Sets the value of the parameter to the `java.sql.Time` value.
`void setTime(int parameterIndex, Time x, Calendar cal)`	Sets the value of the parameter to the `java.sql.Time` value by using the Calendar object.
`void setTimestamp(int parameterIndex, Timestamp x)`	Sets the value of the parameter to the `java.sql.TimeStamp` value.
`void setTimestamp(int parameterIndex, Timestamp x, Calendar cal)`	Sets the value of the parameter to the `java.sql.Timestamp` value by using the calendar object.
`void setURL(int parameterIndex, URL x)`	Sets the value of the parameter to the `java.net.URL` value.

Advantages and Disadvantages of Using a PreparedStatement Object

The advantages of using a `PreparedStatement` object are as follows:

- Improves the performance of an application compared to the `Statement` object that executes the same query multiple times. The `PreparedStatement` object performs the execution of queries faster by avoiding the compilation of queries multiple times.
- Provides a programmatic approach to set the values. In other words, the value of each parameter provided in an SQL query is passed separately by using the `PreparedStatement` object, unlike the `Statement` object.

The disadvantage of using the PreparedStatement object is that it represents only one SQL statement at a time, i.e., you cannot execute more than one statement by a single PreparedStatement.

Using the PreparedStatement Interface

The following are some of the situations when you should use `PreparedStatement` in a JDBC application:

- When a single query is being executed multiple times
- When a query consists of various parameters and complex types (SQL 99 types)

PreparedStatement is used to increase the efficiency and reduce the execution time of a query. An instance of PreparedStatement must be created to execute a precompiled SQL statement. Follow these broad-level steps to use the PreparedStatement interface:

1. Create a PreparedStatement object
2. Provide the values of the PreparedStatement parameters
3. Execute the SQL statements

Let's discuss each of these steps in detail.

Create a PreparedStatement Object

The prepareStatement(String) method of the Connection object is used to create the PreparedStatement object. The Connection object is used to access the PreparedStatement object, where the query supplied in the prepareStatement() method can contain zero or more question marks ('?', known as parameters). The values of question mark parameters can be set after the query is compiled.

The following code snippet shows how to create the PreparedStatement object in a connection:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con= DriverManager.getConnection (url, "user", "password");
String query="insert into mytable values (?,?,?)";
//Step1: Get PreparedStatement object
PreparedStatement ps=con.prepareStatement (query);
```

In the preceding code snippet, the con parameter is the Connection object. This object is used to call the prepareStatement() method to obtain the PreparedStatement object. In the preceding code snippet, ps is the PreparedStatement object created by using the con object.

Provide the Values of the PreparedStatement Parameters

You need to set the values of the question mark placeholders after creating the PreparedStatement object. The values of the question marks can be set by using the setXXX() methods. For example, if the question mark indicates the value of an integer data type, you can use the setInt() method for the particular parameter. If you have a parameter of the Java string, you can call the setString() method to set the value of the parameter. Note that these values should be set before prepared statements are executed.

In PreparedStatement, there is a setXXX() method for each data type declared in Java. The setXXX() method takes two arguments. The first argument indicates the parameter index and the second argument indicates the value of the parameter. Note that the parameter index starts from 1.

The following code snippet shows how to set the values of the question mark parameter:

```
//Step2: setting values for the parameters
ps.setString(1,"abc1");
ps.setInt(2,38);
ps.setDouble(3,158.75);
```

Execute the SQL Statements

You can execute the precompiled SQL statements by using the execute(), executeUpdate(), or executeQuery() method of the PreparedStatement interface. The result of these methods is same as that of the respective methods in the Statement interface.

The following code snippet shows how to execute the SQL statements:

```
//Step3: Executing the SQL statements
int n = ps.executeUpdate(); // n is the number of rows or tables
that are being updated
```

Let us now demonstrate the use of PreparedStatement in an application, where the PreparedStatement object is used to execute the INSERT statement:

```
import java.sql.*;
public class PreparedStatementEx1 {
  public static void main(String s[]) throws Exception {
        Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
        Connection con= DriverManager.getConnection (
```

```
            "jdbc:oracle:thin:@localhost:1521:XE","scott","tiger");
            String query="insert into mytable values (?,?,?)";
            //Step1: Get PreparedStatement
            PreparedStatement ps=con.prepareStatement (query);
            //Step2: set parameters
            ps.setString(1,"abc1");
            ps.setInt(2,38);
            ps.setDouble(3,158.75);
            //Step3: execute the query
            int i=ps.executeUpdate();
            System.out.println("record inserted count:"+i);
            //To execute the query once again
            ps.setString(1,"abc2");
            ps.setInt(2,39);
            ps.setDouble(3,158.75);
            i=ps.executeUpdate();
            System.out.println("query executed for the second time count: "+i);
            con.close();
      }//main
    }//class
```

The preceding code uses the `PreparedStatement` object along with the connection object. The `setXXX()` methods are used to set the values of the arguments. The preceding example sets the values of the integer, string, and double data types. The `executeUpdate()` method is used to retrieve the number of rows affected by executing the SQL statement.

The output of the PreparedStatementEx1.java is shown in Figure 24.12:

```
record inserted count:1
query executed for the second time count: 1
```

Figure 24.12: Displaying the output of the PreparedStatementEx1.java file

The ResultSet Interface

A `ResultSet` interface is an interface provided in the `java.sql` package, and is used to represent data retrieved from a database in a tabular format. It implies that a `ResultSet` object is a table of data returned by executing an SQL query. A `ResultSet` object encapsulates the resultant tabular data obtained when a query is executed. It holds zero or more objects, where each of the objects represents one row that may span over one or more table columns. You can obtain a `ResultSet` object by using the executeQuery or `getResultSet()` method of a `Statement` object. Some of the important points related to a `ResultSet` are as follows:

- ResultSet follows the iterate pattern.
- A ResultSet object is associated with a statement within a connection.
- You can obtain any number of resultsets using one statement; however, only one ResultSet can be opened at a time. When a ResultSet is tried to open by using a statement that is already linked that an opened ResultSet, the existing ResultSet gets closed implicitly.
- ResultSet is automatically closed when its associated statement is closed.

Describing the Methods of ResultSet

The `java.sql.ResultSet` interface provides certain methods to work with the `ResultSet` objects. The methods available in the ResultSet interface are used to move the cursor throughout the ResultSet and read the data.

Table 24.21 describes some of the most commonly used methods in the ResultSet interface:

Table 24.21: Methods of the java.sql.ResultSet Interface

Method	Description
`boolean absolute(int row)`	Moves the cursor to the specified row in the ResultSet object.
`void afterLast()`	Places the cursor just after the last row in the ResultSet object.

Table 24.21: Methods of the java.sql.ResultSet Interface

Method	Description
`void beforeFirst()`	Places the cursor before the first row in the ResultSet object.
`void cancelRowUpdates()`	Cancels all the changes made to the rows in the ResultSet object.
`void clearWarnings()`	Clears all warning messages on a ResultSet object.
`void close()`	Closes the ResultSet object and releases all the JDBC resources connected to it.
`void deleteRow()`	Deletes the specified row from the ResultSet object and the database.
`boolean first()`	Moves the cursor to the first row in the ResultSet object.
`Array getArray(int columnIndex)`	Retrieves the value of the specified column from the ResultSet object.
`InputStream getAsciiStream(int columnIndex)`	Retrieves a specified column in the current row as a stream of ASCII characters.
`Date getDate(int columnIndex)`	Retrieves the column values of the specified types from the current row. The type can be any of the `Java` predefined data types, such as int, long, byte, character, string, double, or large object types.
`Date getDate(String columnLabel)`	Retrieves the specified column from the current row in the ResultSet object. The object retrieved is of the `java.sql.Date` type in the `Java` programming language.
`Date getDate(String columnName, Calendar cal)`	Retrieves the specified column from the current row in the ResultSet object. The object retrieved is of the `java.sql.Date` type.
`int getFetchDirection()`	Specifies the direction (forward or reverse) in which the ResultSet object retrieves the row from a database.
`int getFetchSize()`	Retrieves the size of the associated ResultSet object.
`ResultSetMetaData getMetaData()`	Retrieves the number, type, and properties of the ResultSet object.
`Object getObject(int columnIndex)`	Retrieves a specified column in the current row as an object in the `Java` programming language on the basis of the column index value passed as a parameter.
`Object getObject(int columnIndex, Map <String, Class<?>>map)`	Retrieves a specified column as an object on the basis of the column number and Map instance passed as parameters.
`Object getObject(String columnLabel)`	Retrieves a specified column in the current row as an object on the basis of the column name passed as a parameter.
`Object getObject(String columnLabel, Map<String,Class<?>> map)`	Retrieves a specified column in the current row as an object on the basis of the column name and Map instance passed as parameters.
`int getRow()`	Retrieves the current row number associated with the ResultSet object.
`Statement getStatement()`	Retrieves the Statement object associated with the ResultSet object.
`Time getTime(int columnIndex)`	Retrieves the column values as a `java.sql.Time` object on the basis of column index passed as an integer parameter.
`Time getTime(int columnIndex, Calendar cal)`	Retrieves the column values as a `java.sql.Time` object on the basis of column index as well as the cal object of the Calendar class passed as parameters.
`Time getTime(String columnName)`	Retrieves the column values as a `java.sql.Time` object on the basis of column name passed as a String value.
`Time getTime(String columnName, Calendar cal)`	Retrieves the column values as a `java.sql.Time` object on the basis of String value of column name as well Calendar object cal as parameters.

Table 24.21: Methods of the java.sql.ResultSet Interface

Method	Description
`Timestamp getTimestamp(int columnIndex)`	Retrieves the column values as a `java.sql.Timestamp` object on the basis of the column index passed as a parameter.
`Timestamp getTimestamp(int columnIndex, Calendar cal)`	Retrieves the column values as a `java.sql.Timestamp` object on the basis of the column index and the cal object of the Calendar class passed as parameters.
`Timestamp getTimestamp(String columnName)`	Retrieves the column values as a `java.sql.Timestamp` object on the basis of the column name passed as a parameter.
`Timestamp getTimestamp(String columnName, Calendar cal)`	Retrieves the column values as a `java.sql.Timestamp` object on the basis of the column name and the cal object of the Calendar class passed as arguments.
`int getType()`	Retrieves the type of the ResultSet object used in a connection.
`SQLWarning getWarnings()`	Retrieves the warning reported on the ResultSet object.
`void insertRow()`	Inserts the specified row and content into the ResultSet object and database.
`boolean isAfterLast()`	Specifies whether the cursor of the ResultSet object is at the end of the last row.
`boolean isBeforeFirst()`	Specifies whether the cursor is before the first row in the ResultSet object or not.
`boolean isFirst()`	Specifies whether the cursor is on the first row or not.
`boolean isLast()`	Detects whether the cursor is on the last row of the ResultSet object or not.
`boolean last()`	Moves the cursor to the last row in this ResultSet object.
`void moveToCurrentRow()`	Moves the cursor to the current row in the ResultSet object.
`void moveToInsertRow()`	Moves the cursor to the inserted row in the ResultSet object.
`boolean next()`	Moves the cursor forward one row. The method returns true if the cursor is positioned on a row and false if the cursor is positioned after the last row.
`boolean previous()`	Moves the cursor backward one row. The method returns true if the cursor is positioned on a row and false if the cursor is positioned before the first row.
`void refreshRow()`	Refreshes the current row associated with the ResultSet object with the recent updates.
`boolean relative(int rows)`	Moves the cursor to a relative number of rows or columns specified in the method.
`boolean rowDeleted()`	Retrieves whether the row has already been deleted or not.
`boolean rowInserted()`	Determines whether the current row has an insertion or not.
`boolean rowUpdated()`	Retrieves whether the current row has been updated or not.
`void setFetchDirection (int direction)`	Sets the direction of the ResultSet object.
`void setFetchSize(int rows)`	Sets the size of the ResultSet object.
`void updateArray(int columnIndex, Array x)`	Updates the column in the ResultSet object with a `java.sql.Array` value.
`void updateXXX()`	Updates the column values of the current row of the specified type. The type can be any of the `Java` predefined data types, such as int, long, byte, character, string, double, and the large object types.
`void updateRow()`	Updates the current row with new content.

Table 24.21: Methods of the java.sql.ResultSet Interface	
Method	**Description**
`boolean wasNull()`	Reports whether the last column has a SQL null value or not.
`void updateNull(String columnName)`	Updates a specific column with a NULL value.
`void updateObject(int columnIndex, Object x)`	Updates the specific column with an Object value.
`void updateTime(int columnIndex, Time x)`	Updates the time value with a `java.sql.Time` value.
`void updateTimestamp(int columnIndex, Timestamp x)`	Updates the time value with a `java.sql.Timestamp` value.
`int getConcurrency()`	Retrieves the concurrency mode of the ResultSet object.
`String getCursorName()`	Retrieves the SQL cursor name used by the ResultSet object.

Using ResultSet

After obtaining a ResultSet object, you can use a ResultSet to read the data (ResultSet content) encapsulated in it. Figure 24.13 shows the process flow involved in getting ResultSet from the Statement object and reading the data from the ResultSet object. The st and rs parameters represent the Statement and ResultSet object references, respectively.

Figure 24.13 shows the ResultSet operations:

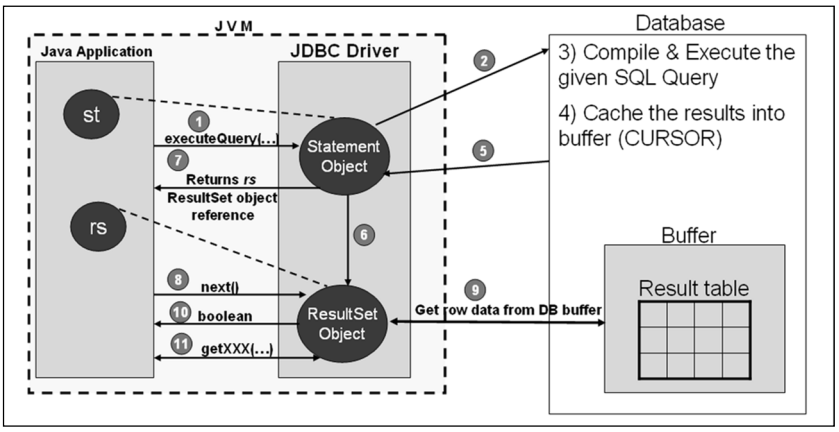


Figure 24.13: Explaining the ResultSet operations

Note that for every `next()` method invoked, the JDBC driver may not necessarily get the data row from the database buffer. This means that after every step 8, shown in Figure 24.13, there may not always be a step 9. Instead, the JDBC driver can get multiple rows of data at a time and buffer it on the client side. The buffering of data on the client side depends on the fetch size set for the ResultSet object. The fetch size of a ResultSet can be set by using the `setFetchSize (int)` method of ResultSet.

You can retrieve data from a ResultSet in two simple steps:

- Move the cursor position to the required row.
- Read the column data using the `getXXX()` methods.

Let's discuss these steps in detail.

Moving the Cursor Position

While obtaining data from a `ResultSet`, the cursor is initially placed before the first row, i.e., `beforeFirst()`. You can use the `next()` method of `ResultSet` to move the cursor position to the next record in the ResultSet. When the cursor is moved to the next record, it returns a Boolean value indicating whether or not any record is available in the ResultSet. The `next()` method returns true if it successfully positions the cursor on the next row; otherwise, it returns false.

NOTE

JDBC 2.0 also introduces some other methods in ResultSet to move the cursor position, provided the ResultSet is of the scrollable type. The ResultSet generated is forwarded by default; therefore, you can iterate through it only in the forward direction from the first to the last row.

Reading the Column Values

After placing the cursor to the specific row, the getter methods of the ResultSet object is used for retrieving the data from that row. Getter methods of the ResultSet object are overloaded, which means that there are two getter methods for each of the JDBC type. One of these two methods takes column index of type `int` as an input, where the column index starts with 1, and the other method takes column name of the String type. You should note that the column names that are passed to the getter methods are not case sensitive. If the same column is present more than once in a selected list, the first instance of the column is to be returned.

Note that the column index supplied to the `getXXX()` methods is the index that starts with 1, where the index numbers are given based on the resulted tabular data and not on the source table that is queried.

For example, suppose a table of students contains two columns: stdId, and stdName. Now, if you obtain a ResultSet object for the select stdName and stdId from the students queries, the column index 1 locates the stdName, while index 2 locates stdId. The ResultSet interface has the `getXXX()` method for all the basic and predefined complex types.

When the getter methods of the ResultSet interface are invoked, the JDBC driver attempts to convert the requested column value into the respective `Java` type and return the `Java` value. However, if it fails to convert the column value into its respective `Java` type, it throws the SQLException exception and describes it as a conversion error.

Figure 24.14 shows the exceptions thrown for the Oracle Thin driver:

```
159     Exception in thread "main" java.sql.SQLException: Fail to convert to int
ernal representation
        at oracle.jdbc.driver.DatabaseError.throwSqlException(DatabaseError.java
:112)
        at oracle.jdbc.driver.DatabaseError.throwSqlException(DatabaseError.java
:146)
        at oracle.jdbc.driver.DatabaseError.throwSqlException(DatabaseError.java
:208)
        at oracle.jdbc.driver.CharCommonAccessor.getInt(CharCommonAccessor.java:
132)
        at oracle.jdbc.driver.OracleResultSetImpl.getInt(OracleResultSetImpl.jav
a:521)
```

Figure 24.14: Showing an example of SQLException

Figure 24.14 shows the error message when the JDBC driver fails to convert the SQL type to the `Java` type. In our case, we have created a Java source file in which the column value is of String type and we have used the `getInt()` method to retrieve the column value.

The ResultSetMetaData Interface

JDBC API provides the feature of MetaData that contains descriptive information about the data available in a database. JDBC MetaData provides information about tables, views, column names, column data types, stored procedures, and databases. In short, we can say that MetaData offers information about the data available in a database as well as all the query results in the database.

JDBC API provides two interfaces, `ResultSetMetaData` and `DatabaseMetaData`, to provide the metadata information of a database. The `ResultSetMetaData` interface offers information about the columns in a ResultSet object, such as the column name, column data type, and length of the columns. The DatabaseMetaData interface provides information about the database structure, table names, views, constraints used in the database, and data types.

Describing the ResultSetMetaData Interface

The `java.sql.ResultSetMetaData` interface is used to describe a JDBC Driver object that encapsulates the metadata of the cursor. The `ResultSetMetaData` object is used to retrieve information about the columns in a ResultSet object. It can also be used to get the information, such as column names, column data types, and column length, related to a particular result. This object does not provide any information about the database or the number of rows present in the result. The `ResultSetMetaData` object is obtained by using the ResultSet object, that is, by invoking the `getMetaData()` method on the ResultSet object. The `ResultSetMetaData` object obtained by using a ResultSet object encapsulates the information about the types and properties of the columns in that ResultSet.

Table 24.22 shows some of the most commonly used methods of the `ResultSetMetaData` interface:

Table 24.22: Methods of the ResultSetMetaData interface

Method	Descriptions
`getColumnCount ()`	Returns an integer value and retrieves the number of columns in the ResultSet object.
`getColumnName (int index)`	Takes the column index and returns the column name.
`getColumnTypeName (int index)`	Takes the column index and returns the database-specific type name.
`getColumnType (int index)`	Takes the column index and returns the integer value that indicates the columns SQL type. It also compares this value with the `java.sql.Types` constants.
`isAutoIncrement (int index)`	Returns the Boolean value indicating whether the column at the given index is auto incremented or not.
`isNullable (int index)`	Indicates whether the column at the given index accepts NULL value or not.
`getCatalogName (int index)`	Accesses the catalog name of the column of a table associated with the ResultSet object.
`getColumnClassName(int index)`	Retrieves the `java` class name associated with the ResultSet object.
`getColumnDisplaySize(int index)`	Retrieves the width of the columns associated with the ResultSet object.
`getColumnLabel(int index)`	Retrieves the title of the specified column in the ResultSet object.
`getPrecision(int column)`	Retrieves the precision value (number of digits after the decimal point) of the decimal digits of the particular column in the ResultSet object.
`getScale(int column)`	Retrieves the number of decimal digits present to the right of the decimal point for a designated column in the ResultSet object.
`getSchemaName(int column)`	Retrieves the schema name of the specified column in the ResultSet object.
`getTableName(int column)`	Retrieves the table name of the specified column in the ResultSet object.
`isCaseSensitive(int column)`	Indicates whether the case of a column matters in context with the associated ResultSet object or not.
`isCurrency(int column)`	Indicates whether the specified column indicates a cash value or not.
`isDefinitelyWritable(int column)`	Indicates that a column is writable.

Table 24.22: Methods of the ResultSetMetaData interface	
Method	**Descriptions**
`isReadOnly(int column)`	Indicates that the specified column is read only and write operations cannot be performed in the column values.
`isSearchable(int column)`	Indicates whether a column value can be used as a condition in the WHERE clause or not.
`isSigned(int column)`	Indicates whether the column values in a particular column are signed or not.
`isWritable(int column)`	Indicates whether a write operation is possible on a column value or not.

After having the theoretical knowledge about the `ResultSetMetaData` interface, let's create an application to demonstrate the implementation of this interface.

Using the ResultSetMetaData Interface

We have learned earlier that the `java`.sql.`ResultSetMetaData` interface provides various methods to retrieve the information about the ResultSet object. The information for the ResultSet object can be retrieved by using the `ResultSetMetaData` object. The following code snippet shows how to create an object of `ResultSetMetaData`:

```
Statement stmt = conn.createStatement();    // Creating a Statement object.
ResultSet rs = stmt.executeQuery();        // creating a ResultSet object.
ResultSetMetaData rsmd = rs.getMetaData(); // Creating  a
  ResultSetMetaData object.
```

The `ResultSetMetaData` object is used to access the metadata for a ResultSet. The following code shows the implementation of the `ResultSetMetaData` interface:

```
import java.sql.*;
import java.util.*;

public class ResultSetMDEx {

  public static void main(String s[]) throws Exception {

        Driver d=(Driver)
        Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
        Properties p= new Properties ();
        p.put ("user", "scott");
        p.put ("password", "tiger");
        Connection con=d.connect ("jdbc:oracle:thin:@192.168.1.123:1521:XE ",p);
        // Creating a Statement object
        Statement st = con.createStatement ();

        ResultSet rs=st.executeQuery ("select * from "+s [0]);
        //Table name is taken as a command line argument

        // Creating the ResultSetMetaData object
        ResultSetMetaData rsmd=rs.getMetaData();

        System.out.println("Table Name : "+s[0]);

        int colcount=rsmd.getColumnCount();

        for (int i=1; i<=colcount; i++) {
                System.out.print (rsmd.getColumnName(i)+"\t");
                System.out.println (rsmd.getColumnTypeName(i));
        }//for
        con.close();
  }//main
}//class
```

The preceding code is used to access the properties of the ResultSet object by using the object of `ResultSetMetaData`. The ResultSet object invokes the `getMetaData()` method, which returns the `ResultSetMetaData` object that is further used to retrieve the column name, types, and length for a specific ResultSet. In ResultSetMDEx, the table for the ResultSet object is given as an argument in the command line. The ResultSetMDEx.`java` application retrieves the columns of the designated table along with the column types and displays it to the user. The user can then put any of the existing tables as a command line argument to retrieve the MetaData information about that table. The output of ResultSetMDEx.java is shown in Figure 24.15(a), 24.15(b), and 24.15(c), which displays various tables, such as mytable, dept, and emp, respectively, used as arguments:

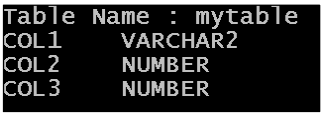


Figure 24.15(a): Showing the fields of table *mytable*

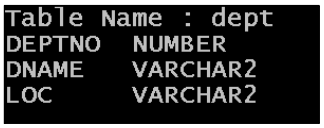


Figure 24.15(b): Showing the fields of table *dept*

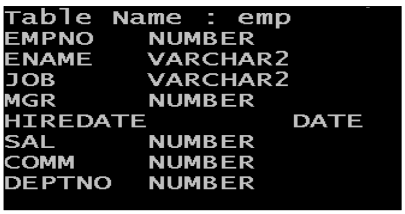


Figure 24.15(c): Showing the fields of table *emp*

Note that the tables for which the metadata is to be displayed must be present in the database to generate a ResultSet object. The following exception, as shown in Figure 24.16, is raised if the table or view (for which the user accesses the metadata) does not exist in the database:

```
Exception in thread "main" java.sql.SQLException: ORA-00942: table or view does
not exist

        at oracle.jdbc.dbaccess.DBError.throwSqlException(DBError.java:134)
        at oracle.jdbc.ttc7.TTIoer.processError(TTIoer.java:289)
        at oracle.jdbc.ttc7.Oall7.receive(Oall7.java:573)
        at oracle.jdbc.ttc7.TTC7Protocol.doOall7(TTC7Protocol.java:1891)
        at oracle.jdbc.ttc7.TTC7Protocol.parseExecuteDescribe(TTC7Protocol.java:
830)
        at oracle.jdbc.driver.OracleStatement.doExecuteQuery(OracleStatement.jav
a:2391)
        at oracle.jdbc.driver.OracleStatement.doExecuteWithTimeout(OracleStateme
nt.java:2672)
        at oracle.jdbc.driver.OracleStatement.executeQuery(OracleStatement.java:
572)
        at ResultSetMDEx.main(ResultSetMDEx.java:17)
```

Figure 24.16: Showing the exception while accessing table that is not available in a database

Figure 24.16 shows the output while accessing a table not present in the database.

Implementing JDBC Processes with **java.sql** Package

The java.sql package is used by a Java application to communicate with a database. The JDBC application-specific code should be written within an application that has to communicate with the database. There are some basic steps to use JDBC in a Java application. Let's now discuss basic steps involved in using JDBC in an application.

Understanding the Basic JDBC Steps

To establish a connection with a database and retrieve the desired results, you need to perform various steps. For example, you need to register a driver with the DriverManager object, obtain a connection, and execute SQL queries.

Figure 24.17 shows the basic steps involved in using JDBC to write a database program in Java:

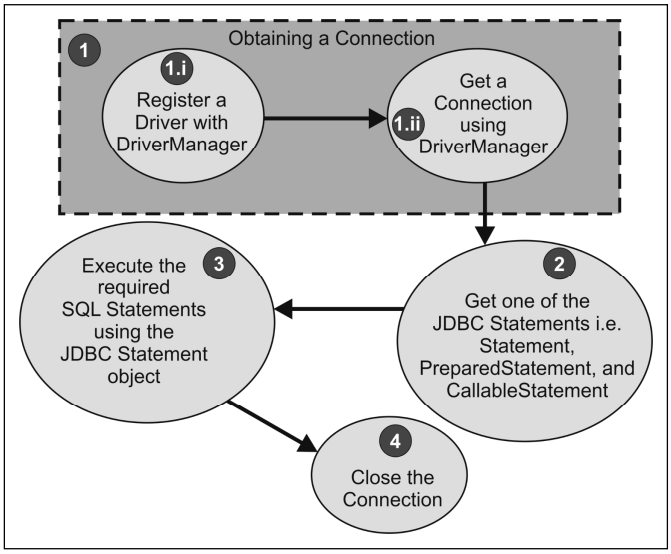


Figure 24.17: Showing basic steps to use JDBC

Figure 24.17 shows the following broad steps that need to be performed to implement JDBC in Java application:

1. Obtaining a connection
2. Creating a JDBC statement object
3. Executing SQL statements
4. Closing the connection

Let's discuss each of them in detail.

Obtaining a Connection

To obtain an object of the Connection interface, you need to first register a driver with the DriverManager class by invoking the `registerDriver()` method, setting the System property, or invoking the `Class.forName()` method. Then the connection is obtained by using the `java.sql.DriverManager` class.

You need to perform the following steps to obtain a connection using the DriverManager class:

a. Register a Driver object with DriverManager.

b. Establish a connection using DriverManager.

Now, let's discuss each of these steps in detail.

Register a Driver object with DriverManager

Registering a driver with the DriverManager class makes the registered driver available to the `DriverManager` class so that the `DriverManager` object can use it to establish a connection with the database. When a driver is registered with the `DriverManager` class, it creates the `DriverInfo` object to maintain the driver details and stores these details in a class variable of the java.util.Vector type.

Establish a Connection Using the DriverManager Class

You can now establish connection with a database after registering the driver with the DriverManager class. To create a connection, invoke any one of the following methods of the DriverManager class:

- getConnection (String url)
- getConnection (String url, String username, String password)
- getConnection (String url, Properties info)

In the preceding methods, <url> is a JDBC URL, which represents a unique name used to identify the driver and obtain the connection. The JDBC URL even contains additional information, such as username and password, required to establish the connection. The syntax of the JDBC URL is as follows:

```
jdbc: <sub protocol> : <info>
```

In the preceding syntax:

- **jdbc**—Represents the protocol in the JDBC URL
- **<sub protocol>**—Specifies the vendor-specific name of the driver used to create the connection
- **<info>**—Takes additional information required to establish the connection, such as the database name and port number, which vary from one driver to another

The following code snippet shows some JDBC URLs:

```
For Type-1 driver, i.e. JDBC-ODBC Bridge Driver, the JDBC URL is:
jdbc: odbc: SuchitaDSN.
For Oracle Type-2 driver:
String dbName = "kogent";
String oracleURL = "jdbc:oracle:oci8:@" + dbName;

//oracleURL = "jdbc:oracle:oci8:@kogent"

For Oracle Type-4 driver:
String host = "localhost";
String dbName = "kogent";
int port = 1521;
String oracleURL = "jdbc:oracle:thin:@" + host + ":" + port + ":" + dbName;
//oracleURL = "jdbc:oracle:thin:@192.168.1.123:1521:XE""
```

When the `getConnection()` method is invoked, it checks if any one of the drivers registered with the `DriverManager` class recognizes the given JDBC URL. If a driver accepts the URL, that driver is used by the `DriverManager` class to establish connection with the DBMS located by the given JDBC URL. Consequently, if no driver accepts the URL, the `DriverManager` class throws the `java.sql.SQLException` exception to the application.

Creating a JDBC Statement Object

You can execute the SQL statements only after creating the JDBC Statement object. The utility objects available to execute SQL statements are `Statement`, `PreparedStatement`, and `CallableStatement`.

Invoke the `createStatement()` method on the current `Connection` object to create the Statement object. The following code snippet shows how to create the Statement object using the `createStatement()` method:

```
Statement stmt = connection.createStatement();
```

Executing SQL Statements

After the `Statement` object is created, it can be used to execute the SQL statements by using the execute(), `executeUpdate()`, or `executeQuery()` methods. The `executeQuery()` method is only used in the SELECT statement. For other database operations, such as INSERT, UPDATE, and DELETE, the `executeUpdate()` method is used to execute statements. The following code snippet shows how to execute an SQL statement:

```
//Using executeQuery()
String query = "SELECT col1, col2, col3 FROM table_name";
ResultSet results = stmt.executeQuery(query);
//Using executeUpdate()
String query= "INSERT into table_name values (value1, value2, …, value n)";
int count = stmt.executeUpdate(query);
```

If the statement produces a ResultSet object after executing the SQL statements, the ResultSet instance is used to retrieve the result. The `next()` method is invoked on the ResultSet object to navigate through a row at a time. The following code snippet shows the use of the ResultSet object within a connection:

```
while(results.next())
{
  System.out.println(results.getString(1) + " " +
  results.getString(2) + " " +
  results.getString(3));
}
```

Closing the Connection

You need to close the connection and release the session after executing all the required SQL statements and obtaining the corresponding results. This can be done by calling the `close()` method of the Connection interface**.** The following code snippet shows how to close a connection:

```
con.close();
```

Processing the ResultSet

We need to set the `ResultSet` type and concurrency to the `Statement` object that produces the `ResultSet` object to work with the `ResultSet` object of the required type. This is done while creating a statement using the `createStatement()`, `prepareStatement()`, and `prepareCall()` methods of the `Connection` object. The following code snippet shows the setting to create `Statement` and `PreparedStatement` objects that can produce a `ResultSet` of the scrollable and read-only type:

```
createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_READ_ONLY);

prepareStatement(query, ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_READ_ONLY);
```

In the preceding code snippet, we have specified the ResultSet type and concurrency while creating the JDBC Statement. Now that we have learned about the various types of `ResultSet` objects, let's look at some examples that demonstrate the working of the types of ResultSet objects by putting the theoretical discussion into action. The example given below demonstrates the working of various ResultSet navigation methods by using scrollable ResultSet:

```
import java.sql.*;
import java.util.*;
import java.io.*;
public class ScrollableRSEx1 {
  public static void main(String s[]) throws Exception {

        Driver d= (Driver) (Class.forName(
         "oracle.jdbc.driver.OracleDriver").newInstance());

        Properties p=new Properties ();
        p.put("user","scott");
        p.put("password","tiger");

        Connection con=d.connect("jdbc:oracle:thin:@localhost:1521:XE",p);
        Statement st= con.createStatement (
        ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);

        String query="select * from emp where deptno=3";
        ResultSet rs= st.executeQuery(query);
        //Now the cursor of ResultSet will at beforeFirst & the result
        set produced is scrollable

        System.out.println("EmpNo\tName");

        while (rs.next()) {
```

```
                System.out.print(rs.getInt(1)+"\t");
                System.out.println(rs.getString(2));
        }//while
        //Now the cursor of ResultSet will at afterLast

        System.out.println(
        "Reading Data, moving the cursor in backward direction\n");

        while (rs.previous()){
                System.out.print(rs.getInt(1)+"\t");
                System.out.println(rs.getString(2));
        }//while
        con.close();
    }//main
}//class
```

In the preceding code, the `Connection.createStatement(int, int)` method is used to create a Statement object that can produce scrollable `ResultSet`. Here we have used `ResultSet.TYPE_SCROLL_INSENSITIVE` to specify the `ResultSet` type, which indicates that `ResultSet` is only scrollable and insensitive to any changes made to the data in the associated database. In addition, we have used `ResultSet.CONCUR_READ_ONLY` to specify the concurrency type, which describes that the `ResultSet` object does not allow you to make changes. It also uses the cursor movement methods, such as `ResultSet.next()` and `ResultSet.previous()`. The `ResultSet.next()` method is used to move the `ResultSet` cursor to the next record in the forward direction, whereas the `ResultSet.previous()` method is used to move the `ResultSet` cursor to the previous record in the backward direction from the current cursor position. The output of ScrollableRSEx1 is shown in Figure 24.18:

```
EmpNo   Name
1015    Puneet
1020    Nikhil
Reading Data, moving the cursor in backward direction

1020    Nikhil
1015    Puneet
```

Figure 24.18: Showing the output of ScrollableRSEx1.java by using ResultSet

The ScrollableRSEx1 shows how to create a `Statement` object that can produce the `TYPE_SCROLL_INSENSITIVE` type of `ResultSet` object. We can also use the `PreparedStatement` object to create the `ResultSet` objects. Let us look at the use of various cursor movement methods as well as `PreparedStatement` to retrieve the scrollable type of the `ResultSet` object:

```
import java.util.*;
import java.io.*;
import java.sql.*;
public class ScrollableRSEx2 {

  public static void main(String s[]) throws Exception {

        Driver d= (Driver)(Class.forName(
        "oracle.jdbc.driver.OracleDriver").newInstance());

        Properties p=new Properties ();
        p.put("user","scott");
        p.put("password","tiger");

        Connection con=d.connect ("jdbc:oracle:thin:@192.168.1.123:1521:XE ",p);

        PreparedStatement ps= con.prepareStatement ("select * from emp",
                ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
        ResultSet rs=ps.executeQuery();
```

```
            System.out.println ("Index\tEmpNo\tName");
            int i=1;
            while (rs.next()){
                    System.out.print ((i++)+"\t");
                    System.out.print (rs.getInt(1)+"\t");
                    System.out.println (rs.getString(2));
            }//while
            System.out.println ();

    //Now the cursor will be at afterLast
    //From this point to move the cursor to 3rd record

            rs.absolute(3);

            System.out.println ("3rd record EmpNo : "+rs.getInt(1));

    /*
    To move the cursor to first record we can use
    rs.first(); or with reference to the current row we can use
    */
            rs.relative(-2);

            System.out.println ("1st record EmpNo: "+rs.getInt(1));
    /*
    To move the cursor to last record use rs.last () or with reference to the
    current row we can use
    */
            rs.afterLast ();    //this moves the cursor to afterLast
            rs.previous ();
    //moves the cursor backward by one row (which in this case moves the
    cursor last row

            System.out.println ("last record EmpNo: "+ rs.getInt (1));
            con.close ();
      }//main
    }//class
```

In the above code, we used the `PreparedStatement` object to retrieve the ResultSet. We have used the `Connection.prepareStatement(String, int, int)` method that specifies the query as select * from emp, `ResultSet` type as `ResultSet.TYPE_SCROLL_SENSITIVE` and `ResultSet` concurrency as `ResultSet.CONCUR_READ_ONLY`. This example also shows the use of the `next ()`, `previous ()`, `afterLast ()`, and `absolute ()` methods to move the cursor throughout the ResultSet. The ResultSet object works according to these methods and retrieves the output according to the specified query. The output of ScrollableRSEx2 is shown in Figure 24.19:

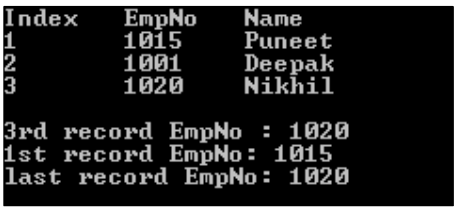


Figure 24.19: Showing the output of the ScrollableRSEx2.java file

In ScrollableRSEx2, we used the `PreparedStatement` object to get the `ResultSet` object of scrollable type. The cursor movement methods are also used to navigate throughout the `ResultSet` object. The `next ()` method is used to retrieve the `ResultSet` data in the forward direction, whose output is shown in Figure 24.19, displaying the records with index 1 to 3. The third record of the `ResultSet` object was retrieved by using the

`ResultSet.absolute(3)` method. The integer argument passed to the method specifies the record position to which the cursor is to be moved. The `ResultSet.relative(int n)` method is used to jump to n rows forward or backward within the `ResultSet`, depending on the sign of n. The negative values allow the cursor to move backward from the current position; however, the positive values move the cursor forward from the current position. If the new position is located before the first row or after the last row of data, false is returned; otherwise, true is returned. In the ScrollableRSEx2, we use `ResultSet.relative(-2)`, where the current cursor position is at third row; therefore, the cursor has moved to the first row. We can use the `ResultSet.last()` method to move the cursor to the last row. However, we used the `ResultSet.afterLast()` method to move the cursor to a position after the last record and then move the cursor one position backward, that is, to the last record using the `ResultSet.previous()` method.

So far we have discussed examples about how to work with different types of ResultSet created by using different `Statement` objects and cursor movement methods. Let's now take an example that shows the difference between using ResultSet types such as `TYPE_SCROLL_SENSITIVE` and `TYPE_SCROLL_INSENSITIVE`. We have learned earlier that the `TYPE_SCROLL_SENSITIVE` type is scrollable as well as sensitive to all the changes, whereas the `TYPE_SCROLL_INSENSITIVE` type is only scrollable. The following code shows the difference in using both the scroll types:

```
import java.sql.*;
import java.util.*;
import java.io.*;
public class ScrollableRSEx3 {
  public static void main(String s[]) throws Exception {
        Driver d= (Driver) (
        Class.forName("oracle.jdbc.driver.OracleDriver").newInstance());

        Properties p=new Properties ();
        p.put("user","scott");
        p.put("password","tiger");

        Connection con1=d.connect("jdbc:oracle:thin:@localhost:1521:XE",p);
        Connection con2=d.connect("jdbc:oracle:thin:@localhost:1521:XE",p);

        Statement st1= con1.createStatement(
                              ResultSet.TYPE_SCROLL_INSENSITIVE,
                              ResultSet.CONCUR_READ_ONLY);
        Statement st2= con2.createStatement();

        String query="select * from emp where deptno=3";

        ResultSet rs= st1.executeQuery(query);
        System.out.println("EmpNo\tName");
        System.out.println("----------------------------------");
        while (rs.next()) {
                System.out.print(rs.getInt(1)+"\t");
                System.out.println(rs.getString(2));
        }//while

        st2.executeUpdate ("update emp set ename=\'kumar\'
                where empno=1020");

        System.out.println("\nData Modified: \nName of the employee
                with empno - 1020 is modified from \'Nikhil\'
                to \'kumar\'\n");

        System.out.println("Now reading the data once again from
                the same ResultSet\n");
```

```
            rs.beforeFirst();
            while (rs.next()){
                    System.out.print(rs.getInt(1)+"\t");
                    System.out.println(rs.getString(2));
            }//while
            con1.close();
            con2.close();
      }//main
    }//class
```

The preceding code uses the `TYPE_SCROLL_INSENSITIVE` type of `ResultSet` object, which indicates that the changes made to the data in the database after obtaining `ResultSet` does not update the `ResultSet` object (Figure 24.19). Figure 24.20 shows that modifying the name of the employee, having EmpNo 1020, from Nikhil to kumar does not modify the records in `ResultSet`:

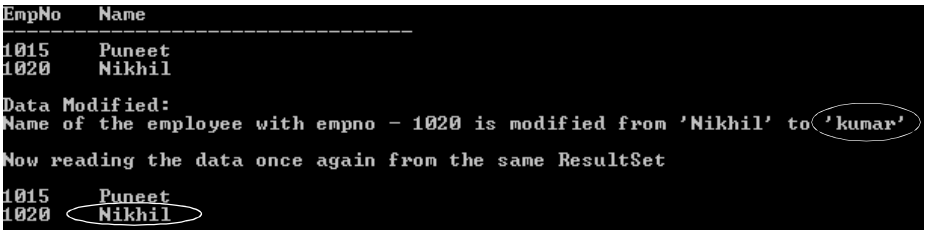


Figure 24.20: Showing the output of the ScrollableRSEx3.java file

Note that in Figure 24.20, even after the data is modified in the underlying data store, it is not reflected in `ResultSet` because in the preceding example, we have configured the `ResultSet` type to `TYPE_SCROLL_INSENSITIVE`.

Now, change the `ResultSet` type in the preceding example to `TYPE_SCROLL_SENSITIVE` by replacing the code snippet shown in boldface in ScrollableRSEx3 with the following code snippet:

```
Statement st1= con1.createStatement (
ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
```

After applying the preceding code snippet, the `ResultSet` object is updated with any changes made to the data store. The output of ScrollableRSEx3 now appears as shown in Figure 24.21:

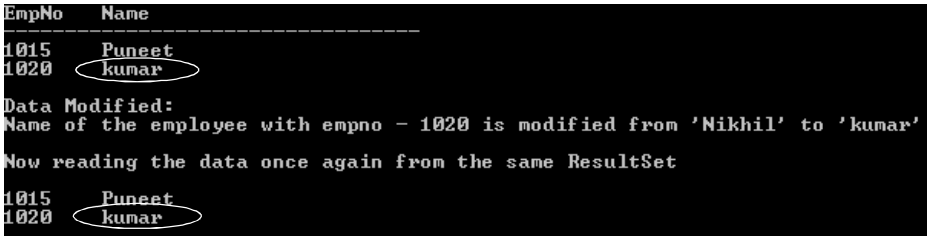


Figure 24.21: Showing the output of ScrollableRSEx3.java using TYPE_SCROLL_SENSITIVE

Figure 24.21 shows that the UPDATE statement changes the existing data in the data store and the changes are reflected in `ResultSet`. The changes are reflected in the `ResultSet` object because the `ResultSet` type `TYPE_SCROLL_SENSITIVE` is used with the application.

Figure 24.20 and Figure 24.21 show that the only difference between the ResultSet of `TYPE_SCROLL_SENSITIVE` and `TYPE_SCROLL_INSENSITIVE` types is that while reading the values from a ResultSet, if the selected data in the underlying data source is modified through some other session, the `TYPE_SCROLL_SENSITIVE` type `ResultSet` reflects the changes, whereas the `TYPE_SCROLL_INSENSITIVE` type does not. Note that if we want the changes to be reflected at any instant with the `TYPE_SCROLL_INSENSITIVE` type of `ResultSet`, we need to explicitly invoke the `refreshRow()` method of `ResultSet` before reading the data in the current row.

NOTE

You can use the TYPE_SCROLL_INSENSITIVE type in the following situations:

- *If the data read in a transaction environment is locked until the transaction is ended*
- *If the data being read is not shared*
- *If we are reading the data that is not required with the latest modifications in an application*
- *If we want to process the data that is available at the time of executing the query*

You can use the TYPE_SCROLL_SENSITIVE type in the following situation.

- *If you need the most recent updated data to execute an application, such as a stock ticker (a stock ticker displays the stock market index of different companies listed in a stock exchange)*

You should remember that selecting an improper `ResultSet` type may result into performance issues and in some cases, the data may get corrupted.

Till now we have learned about the types of `ResultSets` based on the cursor movement and concurrency. The following section explains the ability of `ResultSets` to manipulate the results.

Using the ResultSet Object to Update Data

As discussed earlier, the JDBC ResultSet object supports data manipulation, such as update, delete, and insert. We need to get the `ResultSet` object with a concurrence type as `ResultSet.CONCUR_UPDATABLE` to perform data manipulation. To get an updatable `ResultSet` object, you can create a JDBC `Statement` setting the `ResultSet.CONCUR_UPDATABLE` as discussed earlier in the chapter while understanding scrollable `ResultSet`. The following three steps are involved in using the `ResultSet` object to update the database:

i. Move the cursor to the row where you need to update the data. You can move the cursor by using the ResultSet navigation method as applicable to the ResultSet object.

ii. Use the `ResultSet.updateXXX()` methods to change the data. There are two versions of update methods declared in the `ResultSet` interface. Both these versions of `updateXXX()` methods take two arguments. The first argument is int (specifies the column index - number) for one version and `java.lang.String` (specifies the column name) for the other. The second argument for both the versions is their respective data types that specify a new value, which replaces the old value at the specified column using number or name.

iii. Invoke the `updateRow()` method of the `ResultSet` object. After replacing the data in the row, we need to invoke the `ResultSet.updateRow()` method to reflect the changes into the database.

NOTE

You must note the following points while updating the data by using the ResultSet object:

- *The changes are not updated in the underlying data source until the ResultSet.updateRow() method is invoked.*
- *In order to roll back the changes done to the row, the ResultSet.cancelRowUpdate() method can be invoked. In case the ResultSet.updateRow() method has been invoked already or no updates were made to this row, then this method will have no effect on the data.*
- *The ResultSet.updateRow() method must be invoked without moving the cursor position from the current row.*

While accessing data from ResultSet or iterating through `it` for multiple times, we may want to identify whether the current row was updated using the `ResultSet` object. An application can use the `ResultSet.rowUpdated()` method to find whether the current data is updated or not.

As we have understood how to use the `ResultSet` object to update the database, let's create an example that can put this discussion into action. Let's understand how to update the data using the `ResultSet` object. To understand this, let's display all the employee details and increase the salary of the employeesing who draw more than Rs 2000 by 10%:

```
import java.sql.*;
import java.util.*;
public class IncrementSalary  {

  public static void main(String s[]) throws Exception {
```

```
            Driver d= (Driver) (Class.forName(
            "oracle.jdbc.driver.OracleDriver").newInstance());

            Properties p=new Properties ();
            p.put("user","scott");
            p.put("password","tiger");

            Connection con=d.connect("jdbc:oracle:thin:@localhost:1521:XE",p);

            Statement st=con.createStatement(
                           ResultSet.TYPE_SCROLL_SENSITIVE,
                           ResultSet.CONCUR_UPDATABLE);

            ResultSet rs=st.executeQuery("select empno, ename, sal, deptno
                    from emp");

            System.out.println("Empno\tName\tSalary\tDeptno");
            System.out.println("-----------------------------");

            while (rs.next()) {
                    System.out.print(rs.getInt(1)+"\t");
                    System.out.print(rs.getString(2)+"\t");
                    System.out.print(rs.getDouble(3)+"\t");
                    System.out.print(rs.getInt(4)+"\t");

                    if (rs.getDouble(3)>10000) {
                            double amt=rs.getDouble(3)*1.1;
                            rs.updateDouble(3,amt);
                            rs.updateRow();
                            System.out.print("Salary Incremented");
                    }//if
                    System.out.println ();
            }//while
            con.close();
      }//main
    }//class
```

The preceding code shows how to update the `emp` table in the database by using the ResultSet object. The listing uses the `ResultSet.CONCUR_UPDATABLE` concurrency to create a Statement object so that the queried data can be updated by using the ResultSet object produced by the Statement object. Here, we have selected all the rows of the `emp` table while iterating through the `ResultSet`; the application identifies whether the salary of the employee is greater than 10000, if so, it uses the `updateDouble()` method. The `updateRow()` method is used to commit the changes made to the database by using the ResultSet object. The output of Listing 24.8 is shown in Figure 24.22:

```
Empno   Name    Salary  Deptno
-----------------------------
1015    Puneet  12000.0 3       Salary Incremented
1001    Deepak  12000.0 2       Salary Incremented
1020    kumar   12000.0 3       Salary Incremented
```

Figure 24.22: Showing the output of the IncrementSalary.java file

In Figure 24.22, you can see that the changes made to the employee's salary are indicated as Salary Incremented. The user can view these changes by accessing the database.

Using the ResultSet Object to Delete a Row

You can also delete a row from a database by using the ResultSet object. The deletion of a row from the database indicates that the requested row is completely removed from the database. The following steps must be followed to delete a row by using the ResultSet object:

i. Move the cursor by using any of the cursor movement methods to the row that has to be deleted. Note that the availability of the cursor movement methods depends on the type of ResultSet.

ii. Invoke the `deleteRow()` method on the ResultSet object after moving the cursor to the row that has to be deleted.

After the row is deleted in the underlying data source, the deletion may or may not be visible in the open ResultSet object, depending on the driver implementation. An application can use the `ownDeletesAreVisible(int type)` method of DatabaseMetaData to find whether the deletion is visible as a change or not. The `ownDeletesAreVisible()` method returns true if the deleted row is removed or replaced with an empty row in the ResultSet object of the given type. The ResultSet object returns false if it still has the deleted row. In other words, the deletion of a row is not reflected as a change to the ResultSet object. An application can use the `rs.deleteRow()` method to find whether the current row has been deleted or not. Note that the `rs.deleteRow()` method of the ResultSet object returns true only if the current row was deleted; otherwise, the method returns false.

Let's understand the code to delete a row by using the `deleteRow()` method of the ResultSet object:

```
import java.sql.*;
import java.util.*;
public class RemoveEmployee  {

  public static void main(String s[]) throws Exception {

        Driver d= (Driver) ( Class.forName(
        "oracle.jdbc.driver.OracleDriver").newInstance());

        Properties p=new Properties ();
        p.put("user","scott");
        p.put("password","tiger");

        Connection con=d.connect("jdbc:oracle:thin:@localhost:1521:XE",p);

        Statement st=con.createStatement(
                              ResultSet.TYPE_SCROLL_SENSITIVE,
                              ResultSet.CONCUR_UPDATABLE);

        ResultSet rs= st.executeQuery(
        "select empno, ename, sal, deptno from emp");
        DatabaseMetaData dbmd=con.getMetaData();

        if(dbmd.ownDeletesAreVisible (ResultSet.TYPE_SCROLL_SENSITIVE))
                System.out.println(
                "Deletions are visible as a change for this
                ResultSet object");
        else
                System.out.println(
                "Deletions are not visible as a change for this ResultSet object");

        System.out.println();
        System.out.println("RowNo\tEmpno\tName\tSalary\tDeptno");
        System.out.println("------------------------------");

        while (rs.next()) {
                System.out.print(rs.getRow()+"\t");
                System.out.print(rs.getInt(1)+"\t");
                System.out.print(rs.getString(2)+"\t");
                System.out.print(rs.getDouble(3)+"\t");
                System.out.print(rs.getInt(4)+"\t");

                if (rs.getInt(1)==Integer.parseInt(s[0])) {
                        rs.deleteRow();
                        System.out.print("Employee Removed");
                }//if
                System.out.println();
```

```
            }//while
            con.close();
      }//main
    }//class
```

The code given above removes a record from the database using the ResultSet object. It uses the `ResultSet.deleteRow()` method to delete a specific row using the ResultSet object. Here, we have even used the DatabaseMetaData object to find whether the deletions are visible to ResultSet or not. Figure 24.23 shows the output of RemoveEmployee.java, which deletes a record from the database:

```
Deletions are visible as a change for this ResultSet object

RowNo   Empno   Name    Salary  Deptno
-------------------------------
1       1015    Puneet  13200.0 3       Employee Removed
1       1001    Deepak  13200.0 2
2       1020    kumar   13200.0 3
```

Figure 24.23: Deleting a row using a ResultSet

In Figure 24.23, we can observe a message, Employee Removed for the deleted row, invoking the `ResultSet.deleteRow()` method of the ResultSet object. The `DatabaseMetaData.ownDeletesAreVisible()` method specifies whether the row that is deleted using the ResultSet object is visible or not. In this example, the `DatabaseMetaData.ownDeletesAreVisible()` method has returned true, which means the deletions are visible for ResultSet. When we observe the number of rows in Figure 24.23, it shows that the row number of both the empno 1015 and 1001 is 1. This is due to the deletion of the row with empno 1015. After the row with empno 1001 is removed from ResultSet, the row number of the next row is shifted by one number forward, as the row describing 1015 employee is removed from ResultSet. Therefore, both the records shown in Figure 24.23 have the same row number.

Using a ResultSet Object to Insert a Row

Similar to updating and deleting a row, you can insert a row into the database by using a ResultSet object. The following steps are performed to insert a new row by using the ResultSet object:

i. Move the cursor to the place where you want to insert a row. To do this, you need to invoke the `moveToInsertRow()` method of the ResultSet object.

ii. Set the column value for the new row by using the `updateXXX()` method.

iii. Invoke the `insertRow()` method of the ResultSet object, which inserts a record into the database, after setting all the values.

After inserting a row, you can use the `moveToCurrentRow ()` method to navigate to the previous cursor position.

NOTE

A record is not inserted into the underlying data source until the ResultSet.insertRow() method is invoked. The ResultSet.insertRow() method must be invoked without moving the cursor position from the current row.

As we have learned how to insert a row using a ResultSet object, let's create an example that can demonstrate this. The code given below shows how to insert a new row into a database by using the ResultSet object:

```
import java.sql.*;
import java.util.*;

public class AddEmployee {

  public static void main(String s[]) throws Exception {

        Driver d= (Driver)(Class.forName(
                "oracle.jdbc.driver.OracleDriver").newInstance());

        Properties p=new Properties ();
```

```
            p.put("user","scott");
            p.put("password","tiger");

            Connection con=d.connect("jdbc:oracle:thin:@localhost:1521:XE",p);

            Statement st=con.createStatement(
                    ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);

            ResultSet rs=st.executeQuery("select empno, ename, sal,
            deptno from emp");

            rs.moveToInsertRow();
            rs.updateInt(1,1002);
            rs.updateString(2,"Hari");
            rs.updateDouble(3,13000);
            rs.updateInt(1,3);
            rs.insertRow();
            System.out.print("Employee Inserted");
            con.close();
      }//main
    }//class
```

On execution of the preceding code, an employee record is inserted into the database by using the ResultSet object. The `moveToInsertRow()` method is used to specify the cursor position to insert a new empty row. The `updateXXX()` methods are used to insert the column values for the record to be inserted and then the `insertRow()` method is called by using the ResultSet object to insert the row into the database. Figure 24.24 shows the output of AddEmployee.java:



Figure 24.24: Message Displayed after Inserting a Row

You can also note that the new record is inserted in the database table. The user can retrieve the database explicitly to view the changes made by the ResultSet object.

Let's now learn how to close the connection made to a database.

Closing the Statement

The JDBC Statement object supports the `close()` method to close the statement. The `Statement.close()` method is required to clean the resources of the session allocated for the statement. As described, the closing connection would close the statements built on the Connection object; however, due to the following reasons, you need to explicitly close the Statement:

- You can create only a limited number of Statement on a Connection object; therefore, to create a new Statement, you need to close the unused statements explicitly without closing the underlying connection.
- You know that the statements are closed as soon as their underlying connection is closed. However, this is not properly implemented by all the drivers. Therefore, for maximum portability, we want to explicitly close the statement.

Closing the Connection

You know that the Connection object represents a session established with the server. When a connection is established, some native and network data store resources are allocated for the connection. A user needs to close the connection after completing the required operations to release the allocated resources. JDBC API supports the `Connection.close()` method to close the session represented by the connection. The `close()` method of the Connection object allows us to explicitly release the resources blocked by the connection so that they can be assigned to some other clients, thereby increasing the availability of resources.

The database automatically cleans unused sessions; however, it may take a long time to clear the session. Therefore, using the `Connection.close()` method at proper location in our application is necessary to close the unused session; otherwise, the session represented by the connection would be remained open (active) for a long time without any use.

The database sessions that are identified as unused are automatically closed by the database. The database uses the session timeout to identify the unused sessions. An unused session is the one which is not getting a request for a long time (>session timeout period). Moreover, databases have their own strategies to decide when to find for the unused sessions, such as at regular intervals, or when the numbers of active sessions limit reaches the configured load factor. Therefore, it can be understood that leaving the responsibility of closing the database session to DBMS would block the resources for a long time, which affects the availability of resources and increases the cost of using the system.

NOTE

Opening and closing the database session is a costlier job. Therefore, we need to look for a good design of our application that can utilize the opened connection to the maximum.

The `Connection.close()` method closes the database session associated with a connection that even results to close the Statement and ResultSet (CURSORs) built on this connection. However, with few drivers, the Statement and ResultSet would remain open even after closing the connection. In such a situation, although ResultSet would not be used to read the data, we need to explicitly close the statement and ResultSet. The following section describes closing Statement and ResultSet.

Using a Loop to Set Values in PreparedStatement Object

By using a loop, you can make it easier for you to do coding. You can often make coding easier by using a `for` loop or a `while` loop to set values for input parameters.

In this regard, the code fragment that follows demonstrates using a `for` loop to set values for parameters in the `PreparedStatement` object `updateSales`. This code fragment demonstrates updating the `SALES` column for all the coffees in the table `COFFEES`:

```
PreparedStatement updateSales;
String updateStr = "update COFFEES " +
"set SALES = ? where COF_NAME like ?";
updateSales = con.prepareStatement(updateStr);
int [] salesForWeek = {175, 150, 60, 155, 90};
String [] coffees = {"Colombian", "French_Roast", "Espresso",
"Colombian_Decaf", "French_Roast_Decaf"};
int len = coffees.length;
for(int i = 0; i < len; i++) {
  updateSales.setInt(1, salesForWeek[i]);
  updateSales.setString(2, coffees[i]);
  updateSales.executeUpdate();
}
```

In the preceding code segment, the weekly sales amount is contained in the array `salesForWeek`. These sales amounts correspond to the coffee names specified in the array coffees. The proprietor can easily update the sales amounts for the next week and can use the same code for updating the amounts of coffees. There is no need to change the coffees names.

Returning Values for the executeUpdate() Method

An integer value is returned by the `executeUpdate()` method which specifies the number of updated rows of a table, whereas the `executeQuery()` method returns a `ResultSet` object which contains the results of the query sent to the DBMS. For example, consider the following code snippet that displays the value returned by the `executeUpdate()` method and then assigned to the n variable :

```
updateSales.setInt(1, 25);
updateSales.setString(2, "Espresso");
int n = updateSales.executeUpdate();
```

The table COFFEES gets updated by having the value 25 replaced by the value present in the column SALES in the row for Espresso. This update will affect one row in the table; therefore, the value of n will be equal to 1.

Again, at the time the method executeUpdate() is used to execute a DDL statement, such as in creating a table, it returns the int 0. Consider the following code fragment:

```
int n = executeUpdate(createTableCoffees); // n = 0
```

The preceding code fragment executes the DDL statement which is used for creating the COFFEES table and n is assigned a value of 0.

When the executeUpdate() method returns the 0 value , it can mean one of the following:

- An update statement that affected zero rows was executed or
- A DDL statement was executed.

After going through the program in snippets form, the complete functional program for better understanding is as follows:

```
import java.sql.*;
public class DemoJDBC {
public static void main (String [] args) throws Exception {
      int count = 0;
      double payroll = 0.0;
      ResultSet rslset1=null;
      String names;
      double salaries;
      Driver d= (Driver)(Class.forName(
             "oracle.jdbc.driver.OracleDriver").newInstance());
      Properties p=new Properties ();
      p.put("user","scott");
      p.put("password","tiger");
      Connection con=d.connect(" jdbc:oracle:thin:@localhost:1521:XE",p);
      System.out.println ("*** Connected to the database ***");
      Statement smnt = conn.createStatement ();
      rslset1 = smnt.executeQuery ("select e.Name, e.Salary" + " " +
      "from Employee e" + " " +
      "where e.Dept = 'Toys' ");
      System.out.println ( );
      System.out.println ("Name      Salary");
      System.out.println ("----      ------");
      while(rslset1.next()) {
             names = rslset1.getString (1);
             salaries = rslset1.getInt (2);
             System.out.println (names + "   " + salaries);
             count = count + 1;
             payroll = payroll + salaries; }
      System.out.println ( );
      System.out.println ( );
      System.out.println ("Number of employees in Toys
      Department " + count);
      System.out.println ("Total payroll for Toys
      Department " + payroll);
      smnt.close ();
      PreparedStatement stmt2 = conn.prepareStatement
      ("update Employee"  + " " +     "set Dept=?, Salary=?" + " " +
      "where Name = ? " );
      stmt2.setString(1,"Debit");
      stmt2.setDouble(2,65000.0);
      stmt2.setString(3,"Hari");
      stmt2.executeUpdate();
      stmt2.setString(1,"Credit");
      stmt2.setDouble(2,75000.0);
      stmt2.setString(3,"Leu");
      stmt2.executeUpdate();
```

```
            stmt2.close ();
            conn.commit();
            conn.close ();
      }
   }
```

Using the Interactive SQL Tool

The Novice Programmer appears and says, "There is a need of entering and, at the same time, executing SQL statements. How will people go about it? Is there any answer to the users in this regard?" "Oh well, don't worry Sir", you say. "The Interactive SQL Tool is the answer and will extend support in this regard."

The interactive SQL tool is a means of entering and executing SQL statements. It will be a simple front end to the JDBC API. It will provide a means of entering and executing SQL statements, and at the same time, display areas for viewing results. This tool is basic in terms of functionality, but is useful for experimenting with SQL statements. When you get more familiar with JDBC, extensions can be added to this utility. You'll set the requirement for the Interactive SQL tool class to be fairly simple as follows:

- The first requirement is to enable the user to enter and execute an SQL command.
- The second requirement is to display the `ResultSet` object from an SQL query.
- In addition, the third requirement is to display the error information where ever appropriate.

You'll implement this as an application with a window based on the S`wing` class `Jframe`. A Swing component will also be used by you. It is particularly useful for database application.

Summary

In this chapter, you have learned the basics of JDBC, its architecture, packages, and events. You have gained knowledge on driver interface, the `DriverManager` class. You have also learned to create a basic JDBC program. We have explained the use of PreparedStatement object, interactive SQL tool, and tables.

In the next chapter, you will learn about images and animation.



25

Images and Animation

If you need an immediate solution to:	*See page:*
Using the Image Class	930
Using the MediaTracker Class	937
Using the ImageIcon Class in an Applet	938
Using ImageObserver Interface	942
Creating a Game Using the java.awt.geom Package	943

In Depth

In this chapter, we are going to cover some important topics related to images and animations in detail, along with an understanding of the `java.awt.geom` package. It will also give you knowledge about using `ImageIcon` in an `Applet`. Different stages of animations Will be an added attraction of this chapter.

Images

As its name suggests, the `ImageIcon` class helps to get as well as draw images in Java applets. There are provisions in Java for handling images as Java provides extensive support for imaging. In Java, basic image handling is easy because the Image class in the `java.awt` package provides abstract methods to represent common image behavior. The special methods defined in `Applet` and `Graphics` classes give you everything you need to load and display images in your `applet` as easily as drawing a rectangle.

An image can be referred to as a rectangular graphical object. Images play an important part in Web design. In fact, the breakthrough is the inclusion of the **<**`img`**>** tag in the mosaic browser at the National Center for Supercomputing Applications (NCSA). This is what caused to the Web explosive growth in the year 1993. This tag was used to include an image inline with HTML. Java provides extensive support for imaging because it expands upon this basic concept of allowing images to be managed under program control.

Different Formats of Images

There are different formats of images. The Graphics Interchange Format (GIF) image format was created by CompuServe in the year 1987. Web images could only be in GIF format and could have only up to 256 colors each. This limitation caused the major browser vendors to add support for Joint Photographic Experts Group (JPEG) images. A group of photographic experts created the JPEG format that was used to store full-color-spectrum (includes all wavelengths) and continuous tone images (images that have unlimited range of colors); for example, television. When these images are properly created, they can be compressed more than a GIF of the same source image.

In almost all cases, you'll never care or notice which format is being used in your programs. However, the Java images classes abstract the difference behind a clean interface.

The **ImageIcon** Class

A reference to an icon is generally stored in a variable of type `javax.swing.Icon`. Icon is an interface that declares methods that returns the height and width of an icon in pixels. The `Icon` interface also declares methods to paint the icon on a component by the `component()` method. The class that implements the `Icon` interface is the `javax.swing.ImageIcon`, and this is the class that you use to create an icon object in your program from a file that contains the data defining the icon image.

Several constructors in the `ImageIcon` class accept the icon files in different formats. If it accepts the icon file in Portable Network Graphics format (PNG format), then it will have the `.png` extension, in the case of GIF format (Graphics Interchange Format),`gif` files and; in the case of JPEG format (Joint Photographic Experts Group format), `jpg` files.

The **java.awt.geom** Package

The `java.awt.geom` package plays a very important role in designing images and setting animation effects. It offers Java 2D classes, which in turn define and perform operations on 2D objects. The `AffineTransform` class and the `PathIterator` interface are some of the important classes and interfaces in this package, which is implemented by all `Shape` objects.

The classes that implement the `Shape` interface are `CubicCurve2D`, `Ellipse2D`, `Line2D`, `Rectangle2D`, and `QuadCurve2D`. The mechanisms for operations such as add (union), subtract, intersect, and `exclusiveOR` that are performed on other `Shape` objects are carried out with the help of the `Area` class.

That's it for the overview of this chapter. There's a lot coming up. So it's time to turn to the "Immediate Solutions" section.

Immediate Solutions

Using the **Image** Class

The Novice Programmer appears and says, "What about the different types of operations for images? What is the source for loading images?" You say, "`Image` class provides all the answers.''

While working with images, we come across three common operations, namely, creating an image, loading an image, and displaying an image. The images are taken from the external sources according to the requirements. Thus, Java provides not only ways for you to create a new object and load one but also a means by which an image can be displayed. For example, with Java 1.4, you were able to load and draw image files encoded as JPEG, GIF, and PNG. From Java 5.0 onward, you could also load bitmap formats—BMP and WBMP.

Creating an Image

There are two methods of creating an image. They are discussed as follows:

- `Image createImage (ImageProducer iePr)`—This method returns an image produced by `iePr`. An `ImageProducer` object is present in each image. Its main function is to reconstruct the image whenever needed. For example, reconstructing an image during the modification of its size.
- `Image createImage(int wid, int hgt)`—You can also create an image just to draw on it by using the `createImage()` method from the `Component` class as in the following code:

```
Image image = createImage(width, height);
Graphics g = image.getGraphics();
paint(g);
```

Loading an Image

There are two methods to load an image, which are explained as follows:

- `Image getImage(URL url)`—Refers to the method used for returning an `Image` object that can then be painted on the screen. When an applet draws the image, the data will be loaded on the screen. Then the graphics primitives, which are used for drawing the image, will paint on the screen in an incremental order.
- `Image getImage(URL url, String imgNme)`—Refers to the version which returns an `Image` object that encapsulates the image found at the Uniform Resource Locator (URL) and the name specified by `imgNme`.

TIP

Image is an abstract class, so you are actually loading an instance of a concrete subclass of Image with the getImage() methods above. However, you always treat the object as an instance of Image, i.e., you can only have access to its methods.

To load an image in an application, use:

```
Image im = Toolkit.getDefaultToolkit().getImage(URL or file path);
```

Displaying an Image

An image can be displayed using `drawImage()`. This can be done using two forms:

- `drawImage(Image img, int x, int y, ImageObserver observer)`—It returns Fill with a given image at a given location (upper left corner specified by x and y). Here, `observer` refers to the class that implements the `ImageObserver` interface, which in turn receives the image information notifications as the image is constructed. In other words, it monitors the image while the image is being loaded.
- `drawImage(BufferedImage image, BufferedImageOp op, int x, int y)`—It filters an image. The filter is implemented by the `BufferedImageOp` parameter. The fields and methods of the abstract `Image` class are given in Table 25.1 and Table 25.2 respectively.

Table 25.1: Fields of the Image class	
Field	**Does this**
`protected float accelerationPriority`	Priority for accelerating this image
`static int SCALE_AREA_AVERAGING`	It uses the Area Averaging image-scaling algorithm
`static int SCALE_DEFAULT`	It uses the default image-scaling algorithm
`static int SCALE_FAST`	It chooses an image-scaling algorithm that gives higher priority to scaling speed than smoothness of the scaled image
`static int SCALE_REPLICATE`	It uses the image-scaling algorithm embodied in the `ReplicateScaleFilter` class
`static int SCALE_SMOOTH`	It chooses an image-scaling algorithm that gives higher priority to image smoothness than scaling speed
`static Object UndefinedProperty`	The `UndefinedProperty` object should be returned whenever a property that was not defined for a particular image is fetched

Table 25.2: Methods of the Image class	
Method	**Does this**
`void flush()`	It flushes all reconstructable resources being used by this Image object
`float getAccelerationPriority()`	It returns the current value of the acceleration priority hint
`ImageCapabilities getCapabilities (GraphicsConfiguration gc)`	It returns an `ImageCapabilities` object, which can be inquired as to the capabilities of this Image on the specified `GraphicsConfiguration`
`abstract Graphics getGraphics()`	It creates a graphics context for drawing to an off-screen image
`abstract int getHeight(ImageObserver observer)`	It determines the height of the image
`abstract Object getProperty(String name, ImageObserver observer)`	It gets a property of this image by name
`Image getScaledInstance (int width, int height, int hints)`	It creates a scaled version of this image
`abstract ImageProducer getSource()`	It gets the object that produces the pixels for the image
`abstract int getWidth (ImageObserver observer)`	It determines the width of the image
`void setAccelerationPriority (float priority)`	It sets a hint for this image about how important acceleration is

*Drawing an **Image** Object*

''Hey,'' says the Novice Programmer, ''I was just taking a look at the `Image` object. Do I have to paint images myself?" ''Of course not," you say. ''You can use the `java.awt.image` package to make things a lot easier." ''Tell me all about it," the NP insists.

Images are processed using the streaming framework. This streaming framework involves an image producer, optional image filters, and an image consumer. This framework makes it possible to create an image progressively while it is being taken and generated.

With the help of a framework, applications can discard the storage utilized by an image and regenerate it at any time. The `java.awt.image` package provides a variety of image producers, consumers, and filters that are

configurable to cater the needs of image processing. The interface and classes included in `java.awt.image` package are enlisted in Table 25.3 and Table 25.4.

Table 25.3: The Interfaces of the java.awt.image package

Interface	Does this
`BufferedImageOp`	This interface describes single-input/single-output operations performed on `BufferedImage` objects
`ImageConsumer`	The interface for objects expressing interest in image data through the `ImageProducer` interfaces
`ImageObserver`	An asynchronous update interface for receiving notifications about Image information as the Image is constructed
`ImageProducer`	The interface for objects, which can produce the image data for Images
`RasterOp`	This interface describes single-input/single-output operations performed on `Raster` objects
`RenderedImage`	`RenderedImage` is a common interface for objects, which contain or can produce image data in the form of `Raster`
`TileObserver`	An interface for objects that wish to be informed when tiles of a `WritableRenderedImage` become modifiable by some writer via a call to `getWritableTile()` method, and when they become unmodifiable via the last call to `releaseWritableTile()` method
`WritableRenderedImage`	`WriteableRenderedImage` is a common interface for objects which contain or can produce image data in the form of `Rasters` and can be modified and/or written over

Table 25.4: The classes of the java.awt.image package

Class	Does this
`AffineTransformOp`	This class uses an affine transform to perform a linear mapping from 2D coordinates in the source image or `Raster` to 2D coordinates in the destination image or `Raster`
`AreaAveragingScaleFilter`	An `ImageFilter` class for scaling images by using a simple area-averaging algorithm that produces smoother results than the nearest neighbor algorithm
`BandCombineOp`	This class performs an arbitrary linear combination of the bands in a `Raster`, using a specified matrix
`BandedSampleModel`	This class represents image data which is stored in a band-interleaved fashion and for which each sample of a pixel occupies one data element of the `DataBuffer`
`BufferedImage`	The `BufferedImage` subclass describes an image with an accessible buffer of image data
`BufferedImageFilter`	The `BufferedImageFilter` class subclasses an `ImageFilter` to provide a simple means of using a single-source/single-destination image operator (`BufferedImageOp`) to filter a `BufferedImage` in the `Image Producer/Consumer/Observer` paradigm
`BufferStrategy`	The `BufferStrategy` class represents the mechanism with which to organize complex memory on a particular Canvas or Window
`ByteLookupTable`	This class defines a lookup table object
`ColorConvertOp`	This class performs a pixel-by-pixel color conversion of the data in the source image

Table 25.4: The classes of the java.awt.image package

Class	Does this
`ColorModel`	The `ColorModel` abstract class encapsulates the methods for translating a pixel value to color components (for example, red, green, and blue) and an alpha component
`ComponentColorModel`	A `ColorModel` class that works with pixel values that represent color and alpha information as separate samples and that store each sample in a separate data element
`ComponentSampleModel`	This class represents image data, which is stored such that each sample of a pixel occupies one data element of the `DataBuffer`
`ConvolveOp`	This class implements a convolution from the source to the destination
`CropImageFilter`	An `ImageFilter` class for cropping images
`DataBuffer`	This class exists to wrap one or more data arrays
`DataBufferByte`	This class extends `DataBuffer` and stores data internally as bytes
`DataBufferDouble`	This class extends `DataBuffer` and stores data internally in double form
`DataBufferFloat`	This class extends `DataBuffer` and stores data internally in float form
`DataBufferInt`	This class extends `DataBuffer` and stores data internally as integers
`DataBufferShort`	This class extends `DataBuffer` and stores data internally as shorts
`DataBufferUShort`	This class extends `DataBuffer` and stores data internally as shorts
`DirectColorModel`	The `DirectColorModel` class is a `ColorModel` class that works with pixel values that represent RGB color and alpha information as separate samples and that pack all samples for a single pixel into a single `int`, `short`, or `byte` `quantity`
`FilteredImageSource`	This class is an implementation of the `ImageProducer` interface, which takes an existing image and a filter object and uses them to produce image data for a new filtered version of the original image
`ImageFilter`	This class implements a filter for the set of interface methods that are used to deliver data from an `ImageProducer` to an `ImageConsumer`
`IndexColorModel`	The `IndexColorModel` class is a `ColorModel` class that works with pixel values consisting of a single sample that is an index into a fixed color map in the default RGB color space
`Kernel`	The `Kernel` class defines a matrix that describes how a specified pixel and its surrounding pixels affect the value computed for the pixel's position in the output image of a filtering operation
`LookupOp`	This class implements a lookup operation from the source to the destination
`LookupTable`	This abstract class defines a lookup table object
`MemoryImageSource`	This class is an implementation of the `ImageProducer` interface, which uses an array to produce pixel values for an `Image`

Table 25.4: The classes of the java.awt.image package	
Class	**Does this**
MultiPixelPackedSampleModel	The MultiPixelPackedSampleModel class represents one-banded images and can pack multiple one-sample pixels into one data element
PackedColorModel	The PackedColorModel class is an abstract ColorModel class that works with pixel values which represent color and alpha information as separate samples and which pack all samples for a single pixel into a single int, short, or byte quantity
PixelGrabber	The PixelGrabber class implements an ImageConsumer that can be attached to an Image or ImageProducer object to retrieve a subset of the pixels in that image
PixelInterleavedSampleModel	This class represents image data which is stored in a pixel-interleaved fashion and for which each sample of a pixel occupies one data element of the DataBuffer
Raster	A class representing a rectangular array of pixels
ReplicateScaleFilter	An ImageFilter class for scaling images by using the simplest algorithm
RescaleOp	This class performs a pixel-by-pixel rescaling of the data in the source image by multiplying the sample values for each pixel by a scale factor and then adding an offset
RGBImageFilter	This class provides an easy way to create an ImageFilter, which modifies the pixels of an image in the default RGB ColorModel
SampleModel	This abstract class defines an interface for extracting samples of pixels in an image
ShortLookupTable	This class defines a lookup table object
SinglePixelPackedSampleModel	This class represents pixel data packed such that the N samples, which make up a single pixel, are stored in a single data array element, and each data array element holds samples for only one pixel
VolatileImage	VolatileImage is an image, which can lose its contents at any time due to circumstances beyond the control of the application (e.g. situations caused by the operating system or by other applications)
WritableRaster	This class extends Raster to provide pixel-writing capabilities

To draw an image, use the drawImage() method in the Graphics context:

```
g.drawImage(img, x, y, this);
```

Here, img is a reference to an instance of Image, x and y are the coordinates of the position of the top left corner of the image. The third argument is an ImageObserver reference, which is used internally as a callback to inform a program about the status of image loading. The Component class implements the ImageObserver interface so that you can just put a "this" reference for the argument as default.

The java.awt.Image class is an abstract class. However, methods, such as getImage(URL) in the Applet class, return an object of a concrete subclass which is provided by the JVM implementation. The BufferedImage subclass of Image provides many features to the users accessing it. The Image class provides only minimal access to information about the image. It contains the following methods that return the dimensions of an image:

```
int getWidth(ImageObserver)
int getHeight(ImageObserver)
```

You can draw on an image just as you would draw on a component by obtaining the graphics context object via the `Image` class method:

```
Graphics getGraphics()
```

You then invoke the usual drawing methods in the `Graphics` object to draw on the image.

Loading an Image Object Using the **ImageObserver** Interface

"Is loading an image object as easy as drawing an image object?" asks NP. You say, "That's a tricky part."

The tricky part about images in Java is that they may not load instantly. The `getImage`() method returns the result faster after it is being called so that the program does not hang due to heavy traffic on the Internet line during the slow downloading of the image. In fact, the image loading does not begin until the program attempts to access. For example, drawing of an image.

The problem usually does not arise when loading one or two small- or medium-sized images. It is when several large images are loaded that the problem arises (as in the case of an animation over a slow link).

There are several techniques to monitor image loading and signal when it is fully ready. The fact is that an image is not actually loaded when `getImage()` is invoked. It actually gets loaded when the image is accessed by invoking methods such as `drawImage`() or `getWidth()`. The process of loading is not instantaneous. In fact, it takes a longer time for a slow Internet line. The loading is controlled by an internal thread so that other processes can also continue simultaneously.

The Novice Programmer is satisfied but not fully happy. He says, "Okay, Java helped me to get and draw images. But, what about its updates? I just cannot put an image forever without updating.'' With a smile, you say, "Don't worry, Sir."

When more pixels arrive, you can update the image later. You can create `ImageObservers` too. It overrides the method `imageUpdate()` and examines the loading "manually." If no other steps are taken, then the `drawImage()` method returns immediately, while the image continues to get updated as more number of pixels arrive. The last argument to be passed to `drawImage()` is a reference to an object that implements the `ImageObserver` interface:

```
drawImage (Image img, int x, int y, ImageObserver iob)
```

The thread periodically calls `imageUpdate()` in the `ImageObserver` object to provide status reports on the loading of the image:

```
public boolean imageUpdate (Image img, int info_flags, int x, int y, int w, int h)
```

This method gets the information through the argument info_flags and from the dimension values. When multiple images are loaded, the reference value of the first argument signifies which image is getting updated. When the image loading is not finished yet, the `imageUpdate`() returns true.

We can create an `ImageObserver` class for examining the image loading process, but we usually put the "this" keyword in the last argument, as in `g.drawImage` (`img`, `x`, `y`, `this`), and rely on the default `imageUpdate()` of the `Applet` class. The `Applet` class inherits the methods from the `Component` class and the `Component` class implements this interface and provides a default `imageUpdate()` method.

We can also override `imageUpdate()` and monitor the loading ourselves. The following program `ImageApp` shows one way to implement `imageUpdate()` method. The applet's `start()` method initiates a thread (sub-program) that will timely check a flag—doneloading—that means whether the image has completed loading.

After the `start()` method is called, the `run`() method is invoked next, in which `getWidth(ImageObserver)` initiates the loading process. The AWT provides an image loading machinery that periodically calls back to the `ImageObserver` object that is passed as a parameter in `getWidth()`. It requests for the `imageUpdate()` method and passes the image information.

When `info != ALLBITS`, the method returns true. When `info == ALLBITS`, the flag doneloading is set to true, which signifies that the loop in the `run()` method will terminate and the thread will die. The `imageUpdate()` will then return false and will no longer be invoked. The code is as follows:

```
import javax.swing.*;
import java.awt.*;
/* <APPLET
  CODE=ImageApp.class
  WIDTH=500
  HEIGHT=270 >
</APPLET> */
public class ImageApp extends JApplet implements Runnable {
  DrawPanel drawPanel;
  Image image;
  int imagenum = 0;
  String message ="Loading...";
  boolean doneloading = false;
  public void init () {
        Container conpane = getContentPane ();
        drawPanel = new DrawPanel (this);
        conpane.add (drawPanel);
        image = getImage (getCodeBase (), "logo.jpg" ); }
  public void start () {
        Thread thrd = new Thread (this);
        thrd.start (); }
  public void run () {
        int width = image.getWidth (this);
        if (width >= 0) {
                doneloading = true;
                repaint ();
                return; }
        while (!doneloading) {
                try { Thread.sleep (500); }
        catch (InterruptedException ie) { }
                repaint (); }
  }
  public boolean imageUpdate (Image img, int info, int x,
  int y, int w, int h) {
        if (info != ALLBITS) { return true; }
        else {
                doneloading = true;
                return false; }
  }
}
class DrawPanel extends JPanel {
  ImageApp imgparent = null;
  DrawPanel (ImageApp parent) { imgparent = parent; }
  public void paintComponent (Graphics g) {
  super.paintComponent (g);
        if (imgparent.doneloading) {
                g.drawImage (imgparent.image,10,10,this);
        }
        else { g.drawString (imgparent.message, 10,10); }
  }
}
```

The result of having an `ImageObserver` interface for monitoring the image loading is shown in Figure 25.1:



Figure 25.1: Showing the use of the ImageObserver class

Using the **MediaTracker** Class

"If Java can provide some tools to support image making, will it not be very helpful to all of us?'' asks the Novice Programmer. You say, "They already provide it. There are tools in Java which can control and monitor loading of images."

The `MediaTracker` is useful for loading many images. `MediaTracker` class is used to monitor image loading in a more powerful and elegant way. It provides a method `int status (int ID, boolean load)`, which returns a status indicator in an integer value that is an `OR` of four flags. These four flags are as follows:

- ABORTED
- COMPLETE
- ERRORED
- LOADING

The `statusAll()` method returns a similar value, except that it is an `OR` of the status of all the images currently loaded.

Let's look at this code where an instance of the `MediaTracker` is created and an image is added to it. A thread is initiated to load the image. The `run()` method waits for the tracker to signal that the image is loaded and then the `paint()` method draws an image from drawing of a string message. The code is as follows:

```
import javax.swing.*;
import java.awt.*;
/* <APPLET
  CODE=MediaApp.class
  WIDTH=500
  HEIGHT=270 >
</APPLET> */
public class MediaApp extends JApplet implements Runnable {
  DrawPanel drawPanel;
  MediaTracker Tracker;
  Image image;
  int imagenum = 0;
  boolean show = false;
  String message ="Loading...";
  public void init () {
        Container conpane = getContentPane ();
        drawPanel = new DrawPanel (this);
        conpane.add (drawPanel);
        image = getImage (getCodeBase (), "logo.jpg" );
```

```
            Tracker = new MediaTracker (this);
            Tracker.addImage (image, imagenum); }
      public void start () {
            if (!Tracker.checkID (imagenum) ) {
                  Thread thrd = new Thread (this);
                  thrd.start (); }
             else { show = true; }
      }
      public void run () {
            repaint ();
            try { Tracker.waitForID (imagenum ); }
      catch (InterruptedException e) { }
            if (Tracker.isErrorID (imagenum )){ message= "Error"; }
            else { show = true; }
      repaint (); }
   }
   class DrawPanel extends JPanel {
      MediaApp parent = null;
      DrawPanel (MediaApp parent) { this.parent = parent; }
      public void paintComponent (Graphics g) {
            super.paintComponent (g);
            if (parent.show) { g.drawImage (parent.image,10,10,this); }
            else { g.drawString (parent.message, 10,10); }
      }
   }
```

The result for the code shows how `MediaTracker` monitors image loading (Figure 25.2):



Figure 25.2: Showing the use of the MediaTracker class

Using the **ImageIcon** Class in an Applet

''Hey!" says the Big Boss, ''I was just taking a look at the `Icon` interface. Do I have to paint image icons myself in Swing?" ''Of course not, sir," you say. ''You can use the `ImageIcon` class to make things a lot easier." ''Tell me all about it," the Big Boss says.

The `ImageIcon` class came with J2SE 1.2 in the `javax.swing` package and its purpose was to obtain small icon images on buttons and its importance is that it provides a shortcut technique for loading an image that saves time of the user. `ImageIcon` uses a `MediaTracker` internally, so it provides a convenient way to load a single image as illustrated by this snippet:

```
...
   ImageIcon img = new ImageIcon (url);
   Image image = img.getImage ();
...
```

Basic image handling in Java is easy. The `Image` class in the `java.awt` package provides abstract methods to indicate the common behavior among the images. Special methods defined in `Applet` and `Graphics` provide us everything we need to load and display images in our applet as easily as drawing a triangle. In this section, we'll understand how to get and draw images in our Java applets.

Getting Images

Firstly, we need to load an image in our Java program before it gets displayed. The images are stored as separate files from our Java class files; therefore, we have to indicate Java where to find those files.

The `getImage()` method loads an image and automatically creates an instance of the `Image` class for us. To use it, all we need to do is to import the `java.awt.Image` class into our Java program and then give URL of the image to the `getImage`() method. This can be done in two ways:

- The first one is the `getImage()` method with a single argument; `getImage`(an object of type URL) retrieves the image at that URL.
- The second one is the `getImage()` method with two arguments. The first argument is the base URL (also a `URL` object), while the second is a `String` representing the path or the filename of the actual image (relative to the base).

The first way may appear to be easier (just plug in the `URL` as a `URL` object), and the second way is more flexible. However, as a matter of fact, if hard-coded `URL` of an image is included during the compilation of Java files and then moved to a different location, then all the Java files must be recompiled. The latter form, therefore, is more preferred to use.

The `Applet` class also provides two methods that will help with the base `URL` argument to `getImage()`:

- The first method is the `getDocumentBase()` method: Refers to the method that returns a `URL` object representing the directory of the HTML file containing the applet. Therefore, if the address of the HTML file is `http://www.myserver.com/htmlfiles/javahtml/`, then the `getDocumentBase()` method returns a `URL` pointing to that path.
- The second method is the `getCodeBase()` method: Refers to the method that returns a String representing the directory containing the applet (regardless of the fact that it may or may not be the same directory as the HTML file) and depending on whether the `CODEBASE` attribute in `<APPLET>` is set or not.

The use of either `getDocumentBase()` or `getCodeBase()` depends on whether your images are relative to your HTML files or Java class files. You can use either of them depending upon the prevailing situation or one that suits the situation better.

NOTE

Either of these methods is more flexible to use than write a `URL` or pathname into the getImage() method.

The `getDocumentBase()` or `getCodeBase()` methods are used to move HTML files and applets while a Java can still find your images. This assumes that you move the class files and images around together. If you move the images and class files at different places, then you'll have to edit and recompile your source.

The following form of the `getImage()` method shows that the `image.gif` file and the HTML files are in the same directory that refers to this applet:

```
Image img = getImage(getDocumentBase(), "image.gif");
```

The following form shows that the file `image.gif and the applet itself` are in the same directory:

```
Image img = getImage(getCodeBase(), "image.gif");
```

If you have lots of image files, it's common to put them into their own subdirectory. This type of `getImage()` method resembles the file `image.gif` in the directory images which share the same directory as that of the Java applet:

```
Image img = getImage(getCodeBase(), "images/image.gif");
```

When the `getImage()` method is unable to search the indicated file, it returns null. In case the `drawImage()` `is present` on a null image, it will not draw anything. A probable error will be caused if a null image is used in any other way.

Drawing Images

The `getImage()` method retrieves an image and stuffs it into an instance of the `Image` class. When we are provided with the methods that can load images, we have to display them so as to give an eye-catching content with images as we would do while drawing a rectangle or writing some text. In order to do this, the `Graphics` class has two methods, both of which are known as `drawImage()`.

The `drawImage()` method takes four arguments—the image to be displayed, the `x` and `y` positions of the top left corner, and `this`:

```
public void paint() { g.drawImage(img, 20, 20, this); }
```

This first form draws the image in its original dimensions with the top-left corner at the given **x** and **y** positions. Let's take a look at this code for a very simple applet that loads an image, called `lilies.jpg`, and displays it:

```
import java.awt.Graphics;
import java.awt.Image;
/*<APPLET
   CODE=BugApp.class
   WIDTH=500
   HEIGHT=270 >
</APPLET> */
public class BugApp extends java.applet.Applet {
   Image image;
   public void init() { image = getImage(getCodeBase(), "lilies.jpg"); }
   public void paint(Graphics g) { g.drawImage(image, 10, 10,this); }
}
```

The result of the code appears in Figure 25.3. In this example, the instance variable image holds the lilies image, which is loaded in the `init()` method. The `paint()` method then draws that image on the screen. This example is `BugApp.java`, and you can see its output in Figure 25.3:



Figure 25.3: Drawing an image

The second form of `drawImage()` requires six numbers of arguments, namely, the image to be drawn, the `x` and `y` coordinates of the top-left corner, the width and height of the image bounding box, and `this`. In case the width and height arguments for the bounding box differ in the size from the actual image, then the image will be automatically scaled to fit. Hence, by using those extra arguments, you can compress and expand images according to the space provided for it.

NOTE

Keep in mind, however, that there may be some image degradation or loss from scaling it smaller or larger than its original size.

One useful hint for scaling images is to find the actual size of the image loaded so that you can scale it to a specific percentage and avoid distortion in either direction. The methods that are defined for the `Image` class are `getWidth()` and `getHeight()`. Both these take a single argument, an instance of `ImageObserver`, which is used to track the loading of the image. You'll read more about it later in the chapter. Most of the time, you can use this as a parameter to either `getWidth()` or `getHeight()`.

If you store the logo image in a variable called `bugimg`, for example, this line returns the width of that image, in pixels:

```
theWidth = bugimg.getWidth(this);
```

TIP

If the image isn't loaded fully, you may get different results. Calling getWidth() or getHeight(), before the full loading of the image, will result in the value of -1 for each one. Tracking image loading with ImageObserver can help you keep track when this information appears.

Let's look at another program that uses the `Image` class to display an image in a panel:

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.io.*;
public class PictureFrame extends JFrame implements ActionListener {
  Image img;
  JButton getPictureButton;
  public static void main(String [] args) { new PictureFrame(); }
  public PictureFrame() {
        this.setSize(300, 300);
        this.setTitle("Image Frame");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel picPanel = new PicturePanel();
        this.add(picPanel, BorderLayout.CENTER);
        JPanel buttonPanel = new JPanel();
        getPictureButton = new JButton("Open Image File");
        getPictureButton.addActionListener(this);
        buttonPanel.add(getPictureButton);
        this.add(buttonPanel, BorderLayout.SOUTH);
        this.setVisible(true); }
  public void actionPerformed(ActionEvent e) {
        String file = getImageFile();
        if (file != null) {
                Toolkit kit = Toolkit.getDefaultToolkit();
                img = kit.getImage(file);
                img = img.getScaledInstance(300, -1, Image.SCALE_SMOOTH);
                this.repaint(); }
  }
  private String getImageFile() {
        JFileChooser fc = new JFileChooser();
        fc.setFileFilter(new ImageFilter());
        int result = fc.showOpenDialog(null);
        File file = null;
```

```
            if (result == JFileChooser.APPROVE_OPTION) {
                    file = fc.getSelectedFile();
                    return file.getPath(); }
            else {  return null; }
      }
      private class PicturePanel extends JPanel {
            public void paint(Graphics g) { g.drawImage(img, 0, 0, this); }
      }
      private class ImageFilter
      extends javax.swing.filechooser.FileFilter {
            public boolean accept(File f) {
                        if (f.isDirectory()) { return true; }
                    String name = f.getName();
                    if (name.matches(".*((.jpg)|(.gif)|(.png))"))
                            return true;
                    else {  return false; }
            }
            public String getDescription() {
                    return "Image files (*.jpg, *.gif, *.png)"; }
      }
   }
```

This example uses `JFileChooser` dialog box that lets the user select an image and then display it, (Figure 25.4). In addition to it, the file chooser includes a filter so that only `.jpg`, `.gif` and `.png` files are listed. This example is `PictureFrame.java` and you can view its output in Figure 25.4:

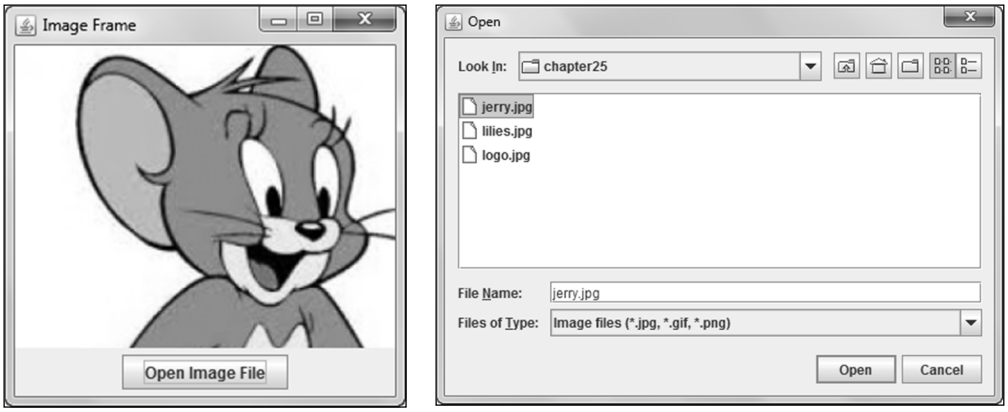


Figure 25.4: The panel showing the chosen image

Using **ImageObserver** Interface

The last argument to `drawImage()`, that is, `this`, is usually not given much importance. This argument also serves as an argument to `getWidth()` and `getHeight()`. This argument is mainly used to pass in an object functioning as an `ImageObserver` (an object that implements the `ImageObserver` interface).

`ImageObserver` plays a very important role. They are used to watch the progress of how far along an image is in the loading process and to make decisions when the image is only fully or partially loaded. Therefore, your applet could pause until all the images are loaded and ready, or display a "loading" message or do something else while it is waiting.

The `Applet` class, which your applet inherits from, contains a default behavior for image observation (which it inherits from the `Component` superclass). Hence, the `this` argument to `drawImage()`, `getWidth()`, and `getHeight()` is used. The only reason you'll want to use an alternate argument in its place is when you want

more control, say in cases where an image may only be partially loaded, or if a lot of image loading has to be tracked asynchronously. Let's have a look at an example:

```
import javax.swing.*;
import java.awt.*;
/* <APPLET
   CODE=App.class
   WIDTH=500
   HEIGHT=200 >
</APPLET> */
public class App extends JApplet {
   public void init () {
         Container conpane = getContentPane();
         Image image = getImage (getCodeBase(), "lilies.jpg");
         DrawPanel drawPanel =  new DrawPanel (image);
         conpane.add (drawPanel); }
}
class DrawPanel extends JPanel {
   Image imag = null;
   DrawPanel (Image image) { imag = image; }
   public void paintComponent (Graphics g)  {
         super.paintComponent (g);
         int imgX = getSize().width/2 - imag.getWidth(this);
         int imgY = getSize().height/2 - imag.getHeight(this);
         g.drawImage(imag, imgX, imgY, this); }
}
```

The result of the code can be seen in Figure 25.5. This example is `App.java` and you can see its output as Figure 25.5:

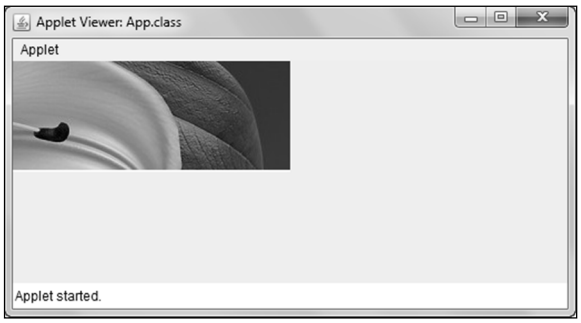


Figure 25.5: Showing the output using the ImageObserver interface

Creating a Game Using the **java.awt.geom** Package

Aniapp is a small animated game, wherein a small kitten runs after the mouse pointer, sleeps, and scratches. For this, you'll implement a small animation based on the kitty graphics. In this gaming application, the kitten appears on the left side of the screen, takes a halt in the middle, yawns, scratches its ear, sleeps a little, and then walks away to the right of the screen. Now, to create the animation, we'll go step by step to understand each and every part of the animation.

Collecting the Images

This is the first step to construct an animation. Prior to developing the code for constructing an animation, you must have all the images that form the animation itself ready. For this version of Aniapp, there are nine of them, as shown in Figure 25.6:



Figure 25.6: Showing the series of figures for animation

Organizing and Loading the Images in the Applet

Once you have constructed an animation, the second step is to organize and load the images in the applet. In this step, an applet plays a role. The basic idea is to show a group of images, one at a time, in a motion depicting that the images are moving. In Java, you can buffer the images in an array of, `Image` class and keep a track of the current image in use with the help of a special variable. During the time of iteration over the slots, the value for the current image can be altered.

In case of the `Aniapp` applet, you must create instance variables. One instance variable is an array that holds the images and another instance variable is of the `Image` type to store the current image that is shown, as done in the following code:

```
Image kittenpics[] = new Image[9];
Image currimg;
```

As the `Aniapp` applet has a total of nine images, the `kittenpics Image` array has nine slots. Therefore, in case of a larger or smaller set of images, there will be different number of slots.

TIP

The java.util package contains a class (Hashtable) that implements a hash table. For larger numbers of images, a hash table is a faster way to find and retrieve images. Since there are a few images here and arrays are better for fixed-length and repeating animation, we'll use an array here.

In the `Aniapp` applet, the kitten images are displayed in different positions on the screen, so you must keep track of the current positions of the `x` and `y` coordinates. This will help the different methods in the applet to be aware of the position from where to start drawing. In this applet, the position for the `y` coordinate always remains constant as the kitten runs from the left side of the screen to the right side at the same `y` position. However, the position for `x` coordinate may vary. The following code shows two instance variables to track x and y positions on the screen:

```
int xposition;
int yposition = 50;
```

While initializing an applet, all the images will be first read and stored in an array. All this initialization process takes place in a method called `init()`.

You can use the `getImage()` method to make an individual call to the nine images having different file names. If you create an array of strings, `kittensrc`, to store the file names and use the `for` loop to iterate the item one by one and load them, it will help in saving time. The following code uses the `init()` method for the `kittensrc` array that loads all the images into the `kittensrc` array:

```
public void init()
{
  String kittensrc[] = { "rt1.gif", "rt2.gif",
        "stp.gif", "yn.gif", "sc1.gif",
        "sc2.gif","slp1.gif", "slp2.gif",
        "awk.gif" };
  for (int i=0; i < kittenpics.length; i++)
  {
        kittenpics[i] = getImage(getCodeBase(),
        "images/" + kittensrc[i]);
  }
}
```

In the preceding code, a call to the `getImage()` method is made. All the images stored in the image directory are added as a part of the path.

Animating the Images

After loading, the next step is animating the images. You do this inside the applet's thread's `run()` method. In this applet, the kitten does five main things that are as follows:

- The kitten runs in from the left side of the screen .
- It stops in the middle and yawns.
- The kitten scratches four times.
- The kitten sleeps.
- The kitten wakes up and runs off to the right side of the screen .

You can easily provide animation effects to this applet by simply painting the right image on the screen at the right time. However, it is advisable to code for this applet in such a way that different activities of the kitten are shown in individual methods. In this manner, you can reuse some of the activities of the kitten, especially the kitten running activity in case you wish the kitten to perform things in a different order.

Let's first make the kitten run by creating a generic method as you will be reusing it twice. For this, the method used is `kittenrun()`, which takes two arguments—the `x` position to start and the `x` position to end. In this case, the kitten runs from the starting x position to the ending x position, `y` remaining constant.

```
void kittenrun(int startposition,int endposition)
{ . . . }
```

To make the kitten run, two images are used, and to bring that running effect, you must alternate between these two images stored at positions 0 and 1 of the image array and move them from one position to another across the screen. To move the kitten, use the `for` loop between the start and end arguments and assign the `x` position with the current loop value. Images can be swapped to check which image is active during the turn of the loop and assign the other image with the current image. Finally, to pause the animation effect at each new frame, you can call the `repaint()` and `sleep()` methods.

While animating the images, several times you will have to pause them at different intervals for a specific amount of time. This can be done by a utility method. The `pause()` method takes a single argument as parameter, that is, number of milliseconds. The code for it is as follows:

```
void pause(int time) {
  try { Thread.sleep(time); }
  catch (InterruptedException e) { }
}
```

Now, let's go back to the `kittenrun()` method. The `kittenrun()` method iterates from the start position to the last position. For each iteration of the loop, it assigns a value to the current `x` position, assigns `currentimg` to the right animation frame, calls the `repaint()` method, and then pauses. The following is the code for the `kittenrun()` method:

```
void kittenrun(int startposition, int endposition) {
  for (int i = startposition; i < endposition; i+=10) {
        xpos = i;
        if (currentimg == kittenpics[0]) { currentimg = kittenpics[1]; }
        else { currentimg = kittenpics[0]; }
  repaint();
  pause(150); }
}
```

In the preceding code, observe the second line of the `kittenrun()` method. The loop is incremented by `10` pixels. Now, you might think why 10 pixels and not 7 or 4? The value to be incremented is decided on the basis of the trial and error method to see what looks right. For this animation, 10 pixel value is the best value to increment. While creating your custom animation, you must do trial and error with both the distance and the sleep time until you come across the desired animation.

Let's now look at the `paint()` method, which is used to paint each frame. In this application, the `paint()` method is quite simple. This method paints the current image at the current `x` and `y` coordinate positions. The instance variables are used to store all such information.

However, before drawing the images, we must first ensure that they exist. To ensure that we are not trying to draw an image that does not exist, it is advisable to first verify whether the `currentimg` variable exists or not prior to calling the `drawImage()` method to paint the image. The code for the `paint()` method is as follows:

```
public void paint(Graphics g) {
  if (currentimg != null) { g.drawImage(currentimg, xposition, yposition, this); }
}
```

Now, let's look at the `run()` method, where the animation's main processing would take place. In the `run()` method, the `kittenrun()` method is called using the appropriate values to make the kitten run from the left side of the screen to the center, as shown in the given code:

```
kittenrun(0, size().width / 2);
```

Apart from running across the screen, the kitten also stops and yawns. A single frame is used for each of these things (in positions 2 and 3 in the array); therefore, you do not really require a separate method to draw them.

Now, you must set the appropriate image by calling the `repaint()` method and pausing the animation for a specific amount of time. In this case, the animation is paused for a second, every time, for stopping and yawning-again by using the trial and error method. The following is the part of code for the same:

```
currentimg = kittenpics[2];
repaint();
pause(1000);
currentimg = kittenpics[3];
repaint();
pause(1000);
```

When the kitten scratches, that is, during the third part of the animation, there seems to be no horizontal movement happening. To show the animation effect, you must alternate between the two scratching images stored at positions 4 and 5 of the image array. As scratching is a separate action, let's create a different method for it.

The `kittenscratch()` method takes a single argument as parameter, that is, the number of times the kitten is made to scratch. This argument will help to iterate through the loop and then alternate between the two scratching images and repaint each time. The following code defines the `kittenscratch()` method:

```
void kittenscratch(int numtimes) {
  for (int i = numtimes; i > 0; i--) {
        currentimg = kittenpics[4];
        repaint(); pause(150);
        currentimg = kittenpics[5];
        repaint();
        pause(150); }
}
```

Within the `run()` method, you can call the `kittenscratch()` method using an argument:

```
kittenscratch(4);
```

The kitten must sleep after scratching. To do so, you can find two images for sleeping in positions 6 and 7 of the array. You can alter these images for a specific number of times. In this case, the `kittensleep()` method takes a single number argument and animates for that many "turns". The following code defines the `kittensleep()` method:

```
void kittensleep(int numtimes) {
  for (int i = numtimes; i > 0; i--) {
        currentimg = kittenpics[6];
        repaint();
        pause(250);
        currentimg = kittenpics[7];
```

```
            repaint();
            pause(250); }
    }
```

The `kittensleep()` method is called in the `run()` method as follows:

```
    kittensleep(5);
```

Finishing Up

Finally, to complete the applet, the kitten wakes up from sleep and then runs off to the right side of the screen. To wake up the kitten, reuse the `kittenrun()` method and use the waking up image, which is the last image in the array at position 8:

```
    currentimg = kittenpics[8];
    repaint();
    pause(500);
    kittenrun(xposition, getSize().width + 10);
```

All the images for this application have white backgrounds, and when you draw these images on an applet, a white box with a medium gray background appears around each image as an applet by default. To solve this issue, simply set the background of the applet, as shown here to white, at the beginning of the `run()` method:

```
    setBackground(Color.white);
```

A lot of code is used in this applet to show the animation effects. The main aim is to set up the frame and then call the `repaint()` method to enable the screen to be redrawn.

One thing to be noted here is that there is nothing done to minimize the amount of flicker in this applet. It appears that the images used in this applet are small and the drawing area is also very small. So, the flicker does not look that bad for this applet.

NOTE

It's always a good practice to develop code for your animation to perform the simplest thing first and then add behavior to make it run smoothly.

The complete code for the applet is as follows:

```
    import java.awt.Graphics;
    import java.awt.Image;
    import java.awt.Color;
    /* <APPLET
      CODE=Kitten.class
      WIDTH=600
      HEIGHT=100 >
    </APPLET> */
    public class Kitten extends java.applet.Applet implements Runnable {
      Image kittenpics[] = new Image[9];
      Image currentimg;
      Thread runner;
      int xposition;
      int yposition = 50;
    public void init() {
      String kittensrc[] = { "rt1.gif", "rt2.gif", "stp.gif",
      "yn.gif", "sc1.gif", "sc2.gif","slp1.gif",
      "slp2.gif",  "awk.gif"  };
            for (int i=0; i < kittenpics.length; i++) {
                    kittenpics[i] = getImage(getDocumentBase(),kittensrc[i]); }
      }
      public void start() {
            if (runner == null) {
                    runner = new Thread(this);
```

```
                runner.start(); }
    }
    public void stop() {
          if (runner != null) { runner = null; }
    }
    public void run() {
          setBackground(Color.white);
          kittenrun(0, getSize().width / 2);
          currentimg = kittenpics[2];
          repaint();
          pause(1000);
          currentimg = kittenpics[3];
          repaint();
          pause(1000);
          kittenscratch(4);
          kittensleep(5);
          currentimg = kittenpics[8];
          repaint();
          pause(500);
          kittenrun(xposition, getSize().width + 10); }
    void kittenrun(int startposition, int endposition) {
          for (int i = startposition; i < endposition; i += 10) {
                xposition = i;
                if (currentimg == kittenpics[0])
                currentimg = kittenpics[1];
                else currentimg = kittenpics[0];
                repaint();
                pause(150); }
    }
    void kittenscratch(int numtimes) {
          for (int i = numtimes; i > 0; i--) {
                currentimg = kittenpics[4];
                repaint();
                pause(150);
                currentimg = kittenpics[5];
                repaint();
                pause(150); }
    }
    void kittensleep(int numtimes) {
          for (int i = numtimes; i > 0; i--) {
                currentimg = kittenpics[6];
                repaint();
                pause(250);
                currentimg = kittenpics[7];
                repaint();
                pause(250); }
    }
    void pause(int time) {
          try { Thread.sleep(time); }
          catch (InterruptedException e) { }
    }
    public void paint(Graphics g) {
          if (currentimg != null) {
                g.drawImage(currentimg, xposition, yposition, this); }
    }
}
```

The result of the code appears in Figure 25.7(a), 25.7(b), 25.7(c), 25.7(d), 25.7(e), and 25.7(f), where the kitten runs, stops, scratches, sleeps, yawns, and again runs, respectively. This example is `Kitten.java` and you can see its output in Figure 25.7:

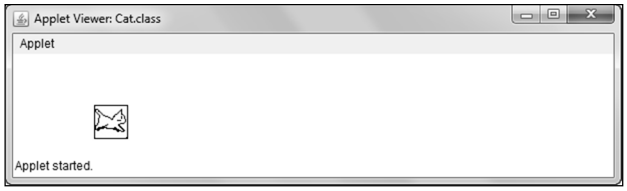


Figure 25.7(a): Kitten runs

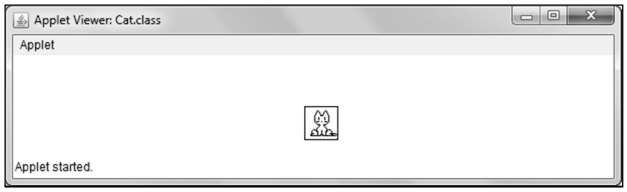


Figure 25.7(b): Kitten stops

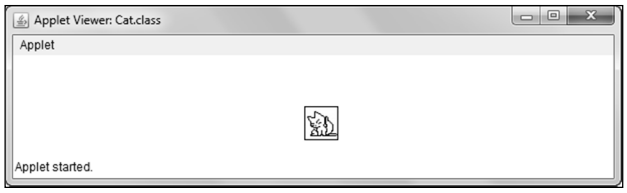


Figure 25.7(c): Kitten scratches



Figure 25.7(d): Kitten sleeps

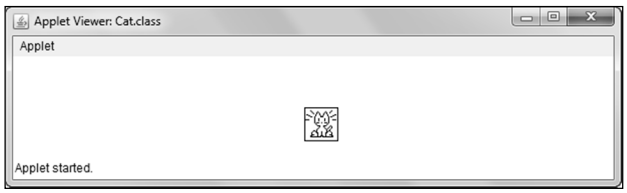


Figure 25.7(e): Kitten yawns

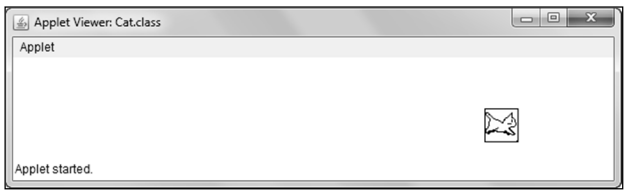


Figure 25.7(f): Kitten again runs

Figure 25.7: Animating Kitten Images

Summary

In this chapter, you have learned the ways to implement images and animations in your Java-based application. You have come across different classes used for implementing the same.

In the next chapter, you'll learn about Java DB.



26

Java DB

If you need an immediate solution to:	*See page:*
Configure Environment Variables	958
Java DB Directory Structure	959
Creating Database in Java DB Using ij-Tool	959
Connecting with a Database	963
Disconnecting from a Database	964
Adding Tables in an Existing Database	964
Deleting Tables from an Existing Database	964
Adding Data into a Table	965
Displaying Content of a Table	965
Java DB Database Encryption	966
Using Java DB with Java Application	966

In Depth

Java DB is the brand name of a database product of Sun Microsystem, which was released with Java 6. Java DB is a lightweight relational database engine based on Java and Structured Query Language (SQL). This is an open-source database technology having Java's write once and run anywhere property. It is a robust, but a cost-effective technology that provides a small footprint. Java DB is easy to deploy and we can migrate an application to other open-source databases using it. You can create applications using Java DB with NetBeans, Eclipse, and Java Studio Creator. Moreover, Java DB has some tools like ij, which can be used with the command line interface.

The standards followed by Java DB are similar to that of the American National Standards Institute (ANSI) SQL or Java Database Connectivity (JDBC). Thus, it has the same Relational Database Management System (RDBMS) functionalities, such as transaction management, concurrency control, and triggers, as the other databases. The syntax of the SQL Query of Java DB is based on SQL92, SQL99, and SQL2003. Java DB provides two environments, namely, multiple databases per system and multiple systems per read-only databases, which make database development simple for developers. Moreover, it provides security using various security mechanisms like database encryption, authentication, and authorization.

Java DB provides the following features:

- **Open source—**Java DB is an open-source technology.
- **Implemented by Java—**Java DB is purely implemented through Java. Thus, it has all the advantages of Java like write once run anywhere and Java Virtual Machine (JVM).
- **Easy to use—**Java DB is an easy to use technology that does not require maintenance. It is a Compact, robust, and mature database technology.
- **Small footprint—**The size of the base package of Java DB is just 2 MB.
- **Standards based—**Java DB follows same standard as JDBC and SQL92, 99, and 2003.
- **Secure—**Java DB provides security against threats through on-disk database encryption, built-in, Lightweight Directory Access Protocol (LDAP), Network Information Service (NIS)+, and pluggable authentication, network client authentication, Derby network client driver, Java Security Manager, and Secure Sockets Layer/ Transport Layer Security (SSL/TLS).

Java DB has a long list of features along with some limitations. Some limitations of the Java DB are as follows:

- Java DB does not support updatable queries with subqueries
- Java DB does have the LIMIT clause
- It has not any built-in clustering solution

Evolution of Java DB

In 1996, Cloudscape Inc., California, founded and started developing a database technology based on Java. Then, in 1997, it released a Java-based RDBMS known as JBMS. The company renamed JBMS as Cloudscape in 1998 and released its new versions in every six months. Informix Software Inc. acquired Cloudscape Inc. in 1999. Now, Cloudscape became the product of Informix Software Inc. In 2001, IBM acquired all databases assets of Informix, including Cloudscape. Cloudscape database engine is now re-branded as IBM Cloudscape. In August 2004, IBM decided to give Cloudscape to the Apache Software Foundation. Apache renamed the Cloudscape project as Apache Derby. Sun joined the Derby project so that it could use Derby as a component of its own products Java. In December 2006, Sun Microsystem released Java 6, which packages Derby with its new name 'Java DB'.

Java DB is not a new database engine; it is exactly the same product as Derby. Eventually, this embedded database engine has three names, that is, Cloudscape, Derby, or Java DB. In this chapter, we proclaim this database software as Java DB.

Table 26.1 presents the history of Java DB:

Table 26.1: History of Java DB

Year	Description
1996	Cloudscape Inc. founded
1997	JBMS released by Cloudscape Inc.
1998	Cloudscape Inc. renamed JBMS as Cloudscape
1999	Cloudscape Inc. acquired by Informix software Inc.
2001	IBM acquired all database-related part of Informix Software Inc.
2004	IBM decided to give Cloudscape to Apache Software Foundation and Apache renamed the Cloudscape project as Apache Derby
2005	Sun announces Java DB
2006	Sun released Java 6, which packages Derby with its new name 'Java DB'.

NOTE

Now that you have learned about the evolution of Java DB, you can differentiate between Java DB and Apache Derby easily; however, still there is a need of separate discussion related to the differences between Derby and Java DB. It is important to note that there is hardly any difference between Java DB and Apache Derby. Derby is a product of Apache, while Java DB is a product of Sun Microsystem. Java DB is exactly the same as Derby. Also, there is no difference between Java DB documentation and Derby documentation. The Java DB 10.6 documentation is similar to Derby 10.6 documentation. Finally, Java DB can be understood as the synonym for Derby and vice versa.

Comparisons between Java DB and MySQL

This section discusses the difference between Java DB and MySQL. This overview can help you in determining when to use Java DB and when to use MySQL. Some of the difference between Java DB and MySQL are as follows:

1. **Operating system supported—**MySQL supports Windows, Linux, UNIX, and Mac OSX. On the other hand, any operating system that can run Java supports Java DB.
2. **License—**MySQL comes with General Public License (GPL), while Java DB comes with Apache License v2 that is totally free.
3. **Maximum database size—**Both of them provide unlimited size of database.
4. **Maximum number of tables—**In both of them you can make unlimited number of table into your database.
5. **Maximum table size—**In MySQL, maximum table size can be 64 TB, but there is no limitation on the size of table with Java DB.
6. **Maximum row and column per table—**In any of them, you can make unlimited number of rows in a table. However, you can make maximum 1000 columns in MySQL and 1012 columns in Java DB.
7. **Interfaces—**MySQL supports many interfaces like Open Database Connectivity (ODBC), JDBC, C, etc., but Java DB only supports JDBC.
8. **Trigger, bind variables, stored procedures, and row-level locking—**Both MySQL and Java DB supports trigger, bind variables, stored procedures, and row-level locking.
9. **Date and Time—**Both are having a broad range of date and time formats.
10. **Online backup—**MySQL supports online backup through InnoDB Hot Backup, Mysqldump, and MySQL Clusters committed log. Java DB, on the other hand, supports online backup through its online backup utility.

Architecture of Java DB

The architecture of Java DB is a simple and solid state-of-the art technology and based on the modular architecture ***Aries***.

Aries stands for Algorithms for Recovery and Isolation Exploiting Semantics. This is actually based on the following three principles:

1. **Write ahead log—**When any change is made on any object, it is first recorded into a log file and then stored in the log file within the stable storage. After saving the log, actual update can be applied on the object.
2. **Log changes during Undo—**On undoing transaction, the log must be updated to prevent the repetition of action with the database.
3. **Repeating history during Redo—**When any crash occurs and the database restarts, Aries checks the history of the database to retrace the actions done before the crash. This activity helps in restoring the system back to its exact state as it was before the crash.

Java DB operates on the following two types of architectures:

1. Embedded architecture
2. Client/server architecture

Embedded Architecture

Java DB documentation refers embedded architecture as embedded mode and it can be used with the embedded Java applications. With this option, Java DB runs in the same JVM as the application.

Figure 26.1 shows the embedded architecture of Java DB:

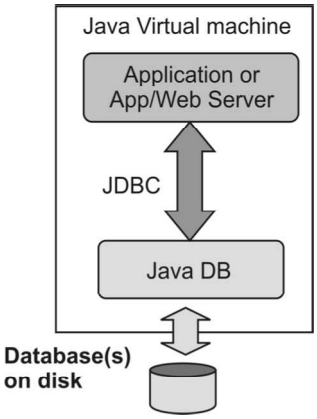


Figure 26.1: Java DB Embedded Architecture

In Figure 26.1, you can see that the Java DB is totally unseen to the end user because Java DB does not require administration of the user and automatically starts and stops by the application. Java DB embedded architecture also provides database reporting and debugging capabilities to applications.

Client/Server Architecture

In the client/server architecture (also known as network-server configuration or client/server mode), several applications can use single database simultaneously. This architecture requires some drivers to use Java DB database. These drivers provide scripts to start and stop the network server. Therefore, client/server architecture requires a little administration.

Figure 26.2 shows the client/server architecture of Java DB:

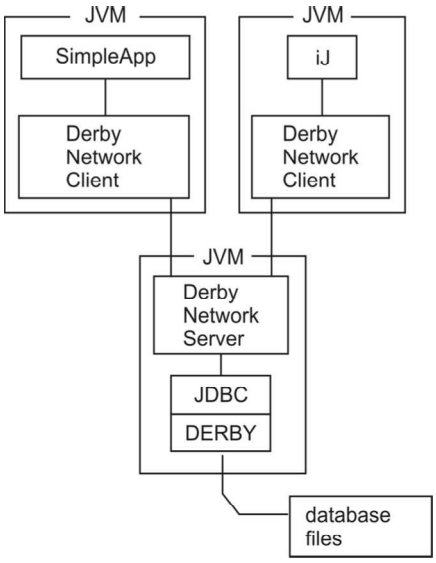


Figure 26.2: Java DB Client/Server Architecture

In Figure 26.2, you can see that Java DB runs in the JVM as embedded as it was in the embedded architecture. JVM hosts to the network server. The client's applications can connect to the network server from different JVMs to access the database. The Java DB network server is a part of the Java DB software distribution. Java DB has the flexibility to support embedded as well as client/server architecture that allows Java DB to be used in a variety of applications.

Compilation of SQL into Java Bytecode

In Figure 26.3, you can see that any SQL query first is compiled into the bytecode through the Query parser and compiler, and then, it can be run on standard JVM. Complying SQL query into bytecode makes the execution of query faster using hotspot compiler. However, it is a burden on the Virtual Machine (VM) because a single VM acts as a database engine for every compilation.

Figure 26.3 shows the process of compilation of the SQL into Java bytecode:



Figure 26.3: Compilation of SQL into Java Bytecode

Drivers of Java DB

A Java DB driver is a program that defines an interface to interact with the Java DB database in a particular mode. Drivers for the Java DB come within the package of Java 6 or the latest version.

It is known that Java DB supports two types of architectures and thus provides the following two types of drivers:

1. **The** `org.apache.derby.jdbc.EmbeddedDriver` **driver—**This driver is used in embedded application when Java DB is running in the same JVM where application is running.
2. **The** `org.apache.derby.jdbc.ClientDriver` **driver—**This driver is used for the network server environment.

Java DB URLs

You need to have Derby JDBC driver to connect with Java DB. Derby JDBC driver is produced by Apache and included in the Java DB distribution package.

Some important features of Derby JDBC driver are as follows:

- It is a Type 4 JDBC driver. Type 4 drivers can use Java networking libraries for communicating with the database server.
- It supports JDBC 4.0 specifications.
- The org.apache.derby.jdbc.EmbeddedDriver class of Derby JDBC driver can be loaded automatically to DriverManager.
- It is available free of cost.

NOTE

Some commercial JDBC-ODBC drivers are also available. For example, Easysoft JDBC-ODBC Bridge.

To connect with an existing Java DB database using the embedded driver, the following syntax is used:

```
jdbc:derby:<Database_Name>;<URLAttrib>
```

where `Database_Name` is an identifier that shows the name of your database to which the embedded driver will connect and URLAttrib specifies some attributes of Uniform Resource Locator (URL), such as create and upgrade, to be used during connection.

To connect to an existing Java DB database using the network client driver, the following syntax is used:

```
jdbc:derby://<Host>:<Port>/<Database_Name>;
```

where Host is an identifier that shows the name computer in which Java DB is running. If Java DB is running on your local system, then use `localhost` or `127.0.0.1` as a Host. Port refers to a port number that can accept incoming requests. `Database_Name` is an identifier that shows the name of the database to which the client wants to connect.

Java DB Product Documentation

Product documentation describes a product to its users. It includes technical manuals, online document, and help manual for the product. The product documentation of Java DB refers to the following manuals:

- **Getting started with Java DB—**This documentation consists of a detail description about the installation and configuration of Java DB. The 'self-study tutorial' included in the documentation is meant for new users and the 'quick-start guide' is meant for experienced users.
- **Java DB developer's Guide—**This documentation consists of a detail description about the features and functionality of Java DB common to all deployments. The documentation includes JDBC specifications, working with properties, deploying Java DB applications, security, etc.
- **Java DB reference manual—**This documentation consists of a detail description about the implementation of SQL language in Java DB. Documentation includes JDBC and Java Transaction API (JTA) implementations, system tables, keywords, and SQLExceptions in Java DB.
- **Tuning Java DB—**This documentation provides performance tips and description of Java DB optimizer.
- **Java DB tools and utilities guide—**This documentation consists of a detail description about the Java DB tools such as ij, sysinfo, and dblook.
- **Java DB server and administration guide—** This document has two parts, the first part consists of the detail description of server configuration, making client applications, and administrating database. The second part contains description about importing and exporting data, administrative tasks such as backing up and restoring databases, storing Java code, and debugging deadlocks.
- **Java DB API reference—** This documentation contains an API reference that is automatically generated for all public Java DB classes.

Derby Tools and Utilities

The Derby tools and utilities are a set of tools and routines available in JavaDB/Derby packaging, which are typically used to manipulate JavaDB/Derby database. The following tools and utilities make the use and development of applications with Java DB easier:

- **The ij scripting tool—**It is an interactive JDBC scripting tool used to run scripts against Java DB database. Through ij commands, you can run your ad hoc queries and access various JDBC features.
- **The import and export utilities—**The import and export utilities are the server-side utilities that allow importing data from files into tables and vice versa.
- **The jar file utilities—**Through the jar file utilities, you can store jar files in a database.
- **The sysinfo utility—**This utility provides you the information of Java DB and its environment.

Installing and Configuring Java DB on Windows

To install Java DB, you must install Java 6 (or any latest version) on your system. Java DB is installed automatically as part of the Java SE Development Kit (JDK). You can download the latest version of JDK from the following link:

```
http://www.oracle.com/technetwork/java/javase/downloads/index.html
```

The installation of Java DB is done in the JDK installation's db directory, wherein scripts and libraries are contained. The subdirectories contained in the installation are as follows:

- Scripts for the execution of utilities and setting up of environment comprise the bin subdirectory
- Java DB jar files comprise the lib subdirectory

Immediate Solutions

Configure Environment Variables

Before the implementation of any Java DB database, you need to configure some environment variables depending on the methods used for running the Java DB tools and startup utilities. The environment variables that you need to set before starting the Java DB server are discussed in the following subsections.

DERBY_HOME

You need to set the DERBY_HOME environment variable to the location of Java DB bin directory. For example, after the installation of Java DB with the JDK 1.8 (Java 8) in Windows, the bin directory of Java DB is created at the location C:\Program Files\Java\jdk1.8.0_31\db. Thus, the DERBY_HOME environment variable is set through the following command:

```
set DERBY_HOME=C:\Program Files\Java\jdk1.8.0_31\db
```

PATH

The `DERBY_HOME\bin` directory is added to the `PATH` environment variable; the Java DB/Derby scripts can be run from any directory through this setting. The `PATH` environment variable should be set with the help of the following command:

```
set PATH=C:\Program Files\Java\jdk1.8.0_31\db\bin;%PATH%
```

NOTE

Once you are through with the setting of the DERBY_HOME environment variable, you can use the following shortened commands:
`set PATH=%DERBY_HOME%\bin;%PATH%`
to add the underlying /bin directory in the PATH environment variable.

It is important to note that the setting of `DERBY_HOME` and `PATH` environment variables is valid only on the current command window; whenever you open a new command window, you must set the `DERBY_HOME` and `PATH` environment variables.

JAVA_HOME

For locating the JVM and Java applications, the scripts used the JAVA_HOME environment variable, as shown in the following command

```
set JAVA_HOME= C:\Program Files\Java\jdk1.8.0_31
```

To verify that the environment variables are set correctly, the `sysinfo` command is used. The information concerning the JVM and the version of Java DB/Derby that you have installed is shown through the `sysinfo` command if there is correct configuration of the environment variables, as shown in Figure 26.4:

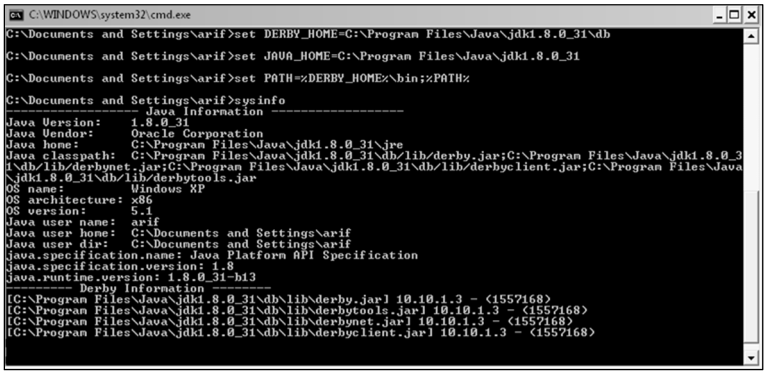


Figure 26.4: Output of the `sysinfo` Command

Java DB Directory Structure

The term directory structure refers to the structured list of files and folders that store the document and files into your computer.

The installation of Java DB creates db directory into JAVA_HOME and also the following subdirectories in the db directory:

1. **lib—**This subdirectory contains the Java DB JAR files. Java DB JAR files help you to start, stop, and access Java DB.
2. **bin—**This subdirectory contains the .exe files for the Java DB.

Creating Database in Java DB Using ij-Tool

It has been discussed earlier that Java DB can run in embedded mode as well as in client/server mode (or Network mode). The creation process of a new database for embedded mode and network mode is different. In the following two subsections, you will learn the creation of database in embedded mode and network mode using ij-tool.

Creating a New Database in Embedded Mode

Java DB package comes with a utility called ij which is used to interact with the Java DB server. Through this tool, you can create, delete, and modify your database. This section describes the creation of a new Java DB database using ij tool.

To create a database with the ij-tool in embedded mode, the following steps are performed:

1. Press the Window + R key from the keyboard.
   The **Run** dialog box appears (Figure 26.5).
2. Type **cmd** in the Open combo box and then click the OK button to open the command prompt, as shown in Figure 26.5:

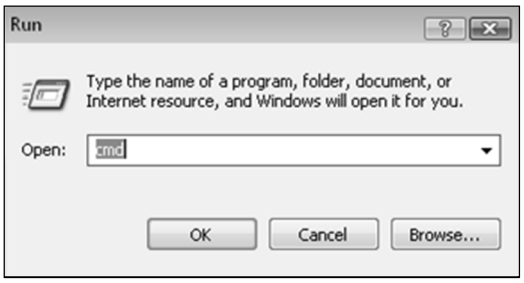


Figure 26.5: Run Dialog Box

The **Command Prompt** appears as shown in Figure 26.6:

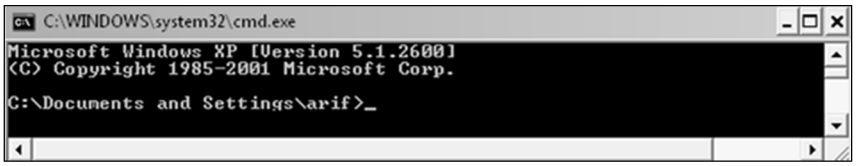


Figure 26.6: Command Prompt

3. Go to the location where you want to create your database. In our case, we have used the `D:\MyJavaDB` location to store and manipulate database, as shown in Figure 26.7:

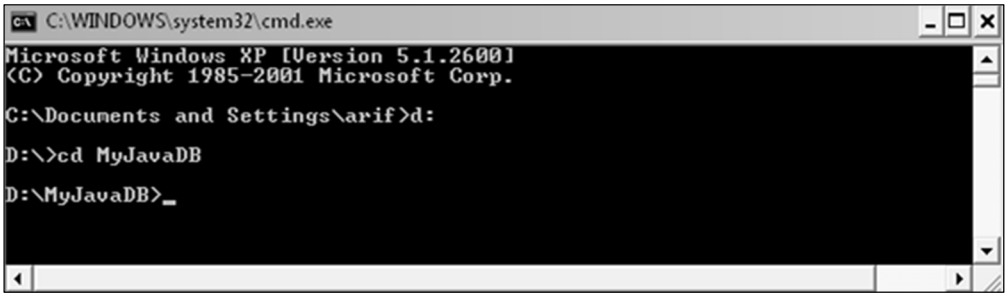


Figure 26.7: Go to the MyJavaDB Folder

4. Type ij and press the Enter key to start the ij-tool, as shown in Figure 26.8:



Figure 26.8: Starting the ij-Tool

When the ij-tool started, you will get an output ij version 10.10 in the **Command Prompt**.

5. Use the `connect` command to create a new database. The syntax of the `connect` command for the embedded mode is as follows:

```
connect 'jdbc:derby:<Database_Name>;create=true;';
```

where `Database_Name` is the name of the database. For example, to create a database `EmpDB` of employee type, use the `connect 'jdbc:derby:EmpDB;create=true;';` command, as shown in Figure 26.9:

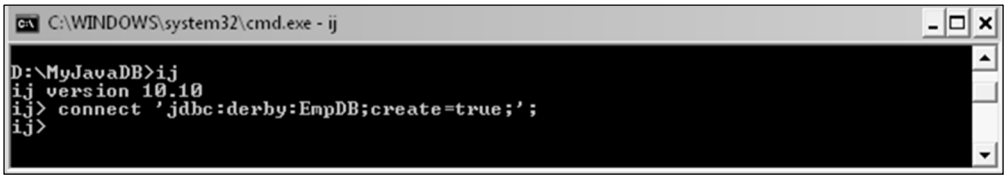


Figure 26.9: Creating the Database `EmpDB`

This command creates a database `EmpDB` (for Employee Database). If you want to protect your database with user name and password, then add user name and password as shown in the following line:

```
connect 'jdbc:derby:EmpDB;create=true;user=scott;password=tiger';
```

This command creates a database into the current directory, i.e., MyJavaDB. Now, you can see a folder in the location where you have created your database, as shown in the Figure 26.10:



Figure 26.10: Database `EmpDB` in the Current Directory

Creating a New Database in Network Mode

It has been discussed in the earlier sections that while creating a new database embedded mode, the embed Java DB can run in the same JVM where the Java application resides. An application running in network mode can access the Java DB database using the client/server method. In this mode, a framework (known as Java DB server or just a server) embeds Java DB database and also handles database requests from other applications (known as Java DB client or just a client). No client can directly interact with the Java DB database. In this section, you will learn to create a Java DB database in network mode using ij-tool.

Starting Network Server

Before the creation of a database, you need to start Java DB server in network mode. Make sure that the environment variables are configured successfully before starting the network server. To start the network server, just type the `startNetworkServer.bat` command into the command prompt and press the Enter key, as shown in Figure 26.11:

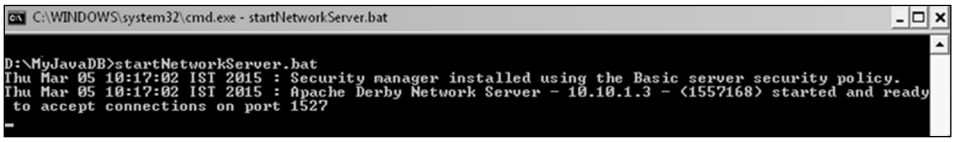


Figure 26.11: Starting Network Server Using the `startNetworkServer.bat` Command

In Figure 26.11, you can see that a message "Apache Derby Network Server – 10.10.1.3 –(1557168) started and ready to accept connections on the port 1527" appears. This message means that the network is successfully started on the port 1527. It is also important to note that the command prompt (like D:\MyJavaDB>) is not present. This is because the window is ready to take the network server processes connection requests, not the DOS command.

NOTE

In Figure 26.10, the current directory is D:\MyJavaDB and the Java DB is started from this directory. So, whenever any database is created on the Java DB server, it will be located in the MyJavaDB directory.

Creating Database

To create a new database in network mode with the ij-tool, use the connect command.

The syntax of the connect command for the network mode is as follows:

```
connect jdbc:derby://<Host>:<Port>/<Database_Name>;create=true;';
```

For example, if you are creating a new database `Network_EmpDB` (for Employee Database) on your local system, then use `localhost` or `127.0.0.0` as a host and `1527` (default) as the port number. Thus, the connection statement can be written as:

```
connect 'jdbc:derby://localhost:1527/Network_EmpDB;create=true;';
```

or

```
connect 'jdbc:derby://127.0.0.0:1527/Network_EmpDB;create=true;';
```

NOTE

Java DB connection URL for Network Server mode is very simple and looks like the embedded mode, that is, just add `//localhost:1527/` *before the database name.*

NOTE

Do you know about the physical location of newly created database? You have studied that there are two types of Java DB drivers, namely, embedded driver (i.e., `jdbc:derby:<Database_Name>;<URLAttrib>;`*) and network client driver (i.e.,* `jdbc:derby://<Host>:<Port>/<Database_Name>;<URLAttrib;`*).*

If you are using the embedded driver during the creation of the database, then the new database will be stored in the same directory in which ij was started up. On the other hand, when you are using the network client driver, then the new database will be stored in the directory where the network server was started up (i.e., DERBY_INSTALL/bin).

Perform the following steps to create a new database:

1. Now start the network server and minimize the window shown in Figure 26.11. In other words, keep the server running.
2. Open a new instance (window) of the Command Prompt and go to the location where you want to create the database (like MyJavaDB directory in our case).
3. Type ij to start the ij-tool in the selected location.
4. Type the following command to create a database Network_EmpDB, as shown in Figure 26.12:

```
connect 'jdbc:derby://127.0.0.0:1527/Network_EmpDB;create=true;';
```



Figure 26.12: Creation of a New Database `Network_EmpDB` in Network Mode Using ij-Tool

Now, a database Network_EmpDB is created into the directory D:\MyJavaDB, as shown in Figure 26.13:

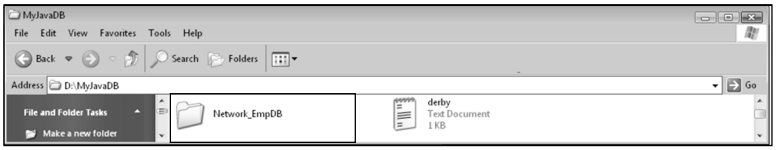


Figure 26.13: Newly Created Database Network_EmpDB in the Directory MyJavaDB

If you want to protect you database with user name and password, then you can add user name and password within the connect statement, as shown in the following line:

```
connect 'jdbc:derby:Network_EmpDB;create=true;user=scott;password=tiger';
```

The preceding command creates a database with specified username and password.

Java DB helps you in specifying the actual location of the database. For example, the following connection URL specifies a full path of a database:

```
connect 'jdbc:derby://localhost:1527//home/MyDB/employee/EmpDB;create=true';
```

Shutting-Down the Network Server

The shutdown process cleans up all engaged database resources and terminates the connection safely. To shut down the network server, perform the following steps:

1. Disconnect from the existing database using the `disconnect;` command. The `disconnect;` command simply disconnects the current database connection (Figure 26.14).
2. Close the ij-tool using the `quit;` command. The `quit;` command lets you out of the ij-tool (Figure 26.14).
3. Stop the network server by using `stopNetworkServer.bat` command. This command stops the network server that is currently running.

   The syntax of the `stopNetworkServer.bat` command is as follows:

```
stopNetworkServer.bat [-h <hostname>] [-p <portnumber>] [-user <username>] [-password
<password>]
```

   where parameters shown in the square brackets are optional.

   As a result of the `stopNetworkServer.bat` command, a message appears, as shown in Figure 26.14:

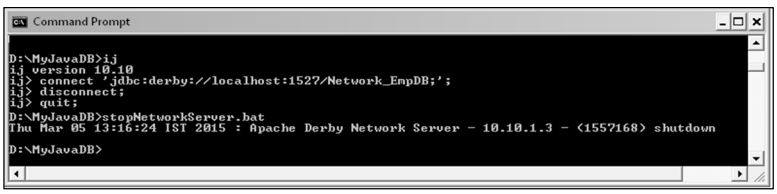


Figure 26.14: Shutdown the Network Server using the `stopNetworkServer.bat` Command

Maximize the window where you have started the network server. In this window, you will also see an echo message about the shutting down process, as shown in Figure 26.15:



Figure 26.15: Echo Massage after Shutting Down the Network Server

Connecting with a Database

Before doing anything with your database, you need to connect with the required database. To connect with an existing database, you have to use the `connect` command.

Connecting with a Database in Embedded Mode

The syntax for making connection with a database running in embedded mode is as follows:

```
connect 'jdbc:derby:<database_name>;';
```

For example, to connect with `EmpDB` in embedded mode, use the following command:

```
connect 'jdbc:derby:EmpDB;';
```

You can find the name of current connection. To display the current connection name, type `show connections;` into the Command Prompt and press Enter.

Figure 26.16 shows the making of connection as well as displaying the name of current connection:



Figure 26.16: Making Connection in Embedded Mode and Displaying the Name of Current Connection

Connecting with a Database in Network Mode

The syntax for making connection with a database running in network mode is as follows:

```
connect jdbc:derby://<Host>:<Port>/<Database_Name>;';
```

For example, to connect with `Network_EmpDB` in network mode, use the following command:

```
connect 'jdbc:derby://localhost:1527/Network_EmpDB;';
```

You can also find the name of current connection. To display the current connection name, type `show connections;` into the Command Prompt and press **Enter**.

Figure 26.17 shows the making of connection as well as displaying the name of current connection of database running in the network mode:



Figure 26.17: Making Connection in Network Mode and Displaying the Name of Current Connection

Note that during creation as well as connection, you need to run the `connect` command; the difference is that the `create=true;` property is used during the creation of a new database, while the `create=true;` property is not used during connection. When you want to create a new database, you have to add the `create=true;` property into the connect statement, which shows that you are giving permission to create a new database. The absence of the `create=true;` property in the `connect` statement denotes the connection with the existing database.

NOTE

If you had applied user name and password during the creation of database, then you must pass user name and password in the `connect` command as:

connect 'jdbc:derby:EmpDB;user=scott;password=tiger';

Disconnecting from a Database

After the completion of manipulation of Java DB database running in either embedded or network mode, you need to properly disconnect your database. To disconnect from the current database running in either modes, type `disconnect;` in the Command Prompt and press Enter.

Figure 26.18 shows disconnection from the current database running in embedded mode and network mode, respectively.



(a) Embedded Mode **(b) Network Mode**

Figure 26.18: Disconnect from the Current Database in Various Modes

Adding Tables in an Existing Database

You can create a new table into an existing database running in either embedded mode or network mode by using the `create table` statement.

The syntax for the `create table` statement is as follows:

```
Create table <table_name> (     <colm_name1> <datatype>,
                                <colm_name2> <datatype>,
                                ...
                                <colm_nameN> <datatype>
                                );
```

where `table_name` is the name of your table, `colm_name1, colm_name2, …, colm_nameN` are the name of the columns, and `datatype` is the data type of column.

For example, to create a table (EmpRecord in our case) for storing employee's record, type the following statement and make sure that the connection with the database (EmpDB in our case) has been stabilized:

```
create table EmpRecord    (empid int not null,
                           empname varchar(30),
                           empdesig varchar(30),
                           empdepart varchar(30),
                           empsalary int);
```

When the table is created successfully, the following message echoed out on the screen:

```
 > > > > 0 rows inserted/updated/deleted
```

Deleting Tables from an Existing Database

You can delete any table from the database running either in embedded mode or in network mode. The `drop table` statement is used to delete a table. The syntax of the `drop table` statement is:

```
Drop table <Table_Name>;
```

For example, if you want to delete the `EmpRecord` table, then type the following statement and press Enter:

```
Drop table EmpRecord;
```

After successful deletion, you will get the following message:

```
> > > > 0 rows inserted/updated/deleted
```

Adding Data into a Table

The `insert into` statement is used to insert a record in a table of a database running in either embedded mode of network mode. There are two forms of the `insert into` statement.

In the first form, you need not specify the column names where the values will be inserted:

```
Insert into <table_name> values (<value1>, <value2>, …, <valueN>);
```

However, in the second form, you must specify both the column names and their values:

```
Insert into <table_name> (<column1>, <column2>, …, <columnN>) values (<value1>, <value2>,
  …, <valueN>);
```

Let's insert some records (or rows) into the EmpRecord table of a database EmpDB running in embedded mode. To insert record, type the following statements one by one and press Enter:

```
insert into EmpRecord values (1,'JOHNSON','MANAGER','ADMIN',48000);
insert into EmpRecord values (2,'HARDING','SALES MANAGER','ADMIN',45000);
insert into EmpRecord values (3,'TAFT','SALES EXEC','SALES',25000);
insert into EmpRecord values (4,'HOOVER','SALES EXEC','SALES',27000);
insert into EmpRecord values (5,'LINCOLN','SALES EXEC','SALES',22000);
```

Figure 26.19 shows the insertion of five rows into the EmpRecord table of the EmpDB database:

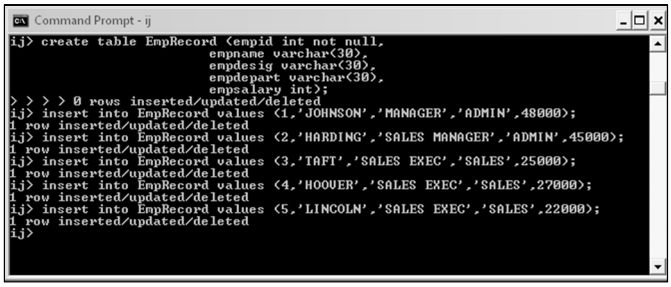


Figure 26.19: Entering Five Rows into the `EmpRecord` Table

In the preceding figure, you can see the message "`1 row inserted/updated/deleted`" echo out on the screen after every successful insertion of the row.

Displaying Content of a Table

The `Select` statement is used to display the content of the table of a database running either in the embedded mode or network mode. The resultant table is known as **result-set**. The syntax of the `Select` statement is:

```
Select <colm_name1>, <colm_name2>, …,<colm_nameN> from <table_name>;
```

This statement selects the result-set having specified set of columns like `<colm_name1>, <colm_name2>, …, <colm_nameN>`. Another form of the `select` statement is as follows:

```
Select * from <table_name>;
```

This statement is used to select all the columns of the table. Here, * is used for 'everything'.

For example, when we pass the `select * from EmpRecord;` query, it will select and display all the content of the `EmpRecord` table, as shown in Figure 26.20:

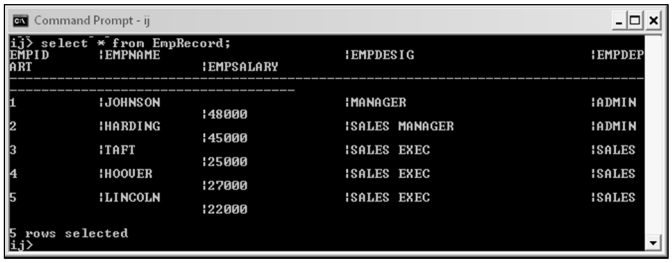


Figure 26.20: Result of the `select * from EmpRecord;` Query

Java DB Database Encryption

A Java DB is just a directory consisting of transaction logs, data files, and a `service.properties` file. Anyone can copy Java DB directory and boot it using another Java DB instance and get the content of the database.

To prevent unauthorized access, you can encrypt your database during creation using a password. The encrypted data can be accessed only through the decryption of the database using the same password as it was used during encryption.

The syntax of encryption of a database running in the embedded mode is as follows:

```
connect  'jdbc:derby: <database_name; create=true;
dataEncryption=true; bootPassword=<your_password>;
encryptionAlgorithm=Blowfish/CBC/NoPadding;
user=<user_name>';
```

Here, `user_name` is the database owner. It is not compulsory to specify a user name.

The syntax of encryption of a database running in a network mode is as follows:

```
connect  'jdbc:derby://<Host>:<Port>/<Database_Name>;create=true;
dataEncryption=true; bootPassword=<your_password>;
encryptionAlgorithm=Blowfish/CBC/NoPadding;
user=<user_name>';
```

The preceding statements have the following three attributes inside the connection string:

- `dataEncryption`—This attribute tells the Java DB to create an encrypted database.
- `bootPassword`—Through the given password, Java DB encrypts the database files. The password length must be more than or equal to 16 characters because Blowfish is a 128-bit encryption algorithm.
- `encryptionAlgorithm`—This is the algorithm used to encrypt the database.

For example, the following `connect` statement creates an encrypted database `TestDB` in embedded mode:

```
connect  'jdbc:derby:TestDB; create=true;
         dataEncryption=true; bootPassword=mypassword;
         encryptionAlgorithm=Blowfish/CBC/NoPadding;
         user=scott';
```

The preceding command creates an encrypted database `TestDB` with the password `mypassword`, where the encryption algorithm is `Blowfish` and the owner of this database is `scott`. Now, we have an encrypted database and we need to connect to this database from our Java application. A part of code that helps you to connect your java application with the encrypted database is as follows:

```
  ...
class.forName("org.apache.derby.jdbc.ClientDriver").newInstance();
    Connection connection = DriverManager.getConnection
                  ("jdbc:derby:<database_name>;bootPassword=<your_password>");
    ...
```

You just need to add your bootPassword in the connection string. If you try to connect to the database without or with the wrong bootPassword, then you will get an error message.

Using Java DB with Java Application

In the preceding sections, you have studied about working with the Java DB database with ij tool. In this section, you will learn about working with the Java DB database with your Java application.

To create a Java DB database with a Java program, you need to call a connection object. The syntax of creating database in Java is as follows:

```
DriverManager.getConnection("jdbc:derby:<database_name>;create=true");
```

This statement causes the Java DB to create the database database_name in the current directory.

You can use the following code to create a new database (Emp_DB in our case):

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class DBConnect {
  public static void main(String[] args)        {
```

```
    try   {
          //String driver = "org.apache.derby.jdbc.EmbeddedDriver";
Connection con = DriverManager.getConnection("jdbc:derby:Emp_DB;create=true;");
          System.out.println("Emp_DB created!");
          }
    catch (Exception err) {
                              System.out.println(err.getMessage());
                              }
    }
}
```

You need to close the connection of database explicitly. To shutdown Java DB, you must use the connection object to initiate the shutdown sequence:

```
DriverManager.getConnection("jdbc:derby:<database_name>;shutdown=true");
```

NOTE

Once the db is running, don't stop it until the application is shut down.

The following Java program shows a complete example of Java DB database:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSetMetaData;
public class DBConnect{
private static String dbURL =
"jdbc:derby://localhost:1527/Student_DB;user=scott;password=tiger;";
private static String tableName = "students";
private static Connection conn = null;
private static Statement stmt = null;
public static void main(String[] args){
createConnection();
insertStudents(1, "Suchita", "Jain");
selectStudents();
shutdown();
}
//Establishing Connection
private static void createConnection(){
try {
Class.forName("org.apache.derby.jdbc.ClientDriver").newInstance();
//Get a connection
conn = DriverManager.getConnection(dbURL);
        }
        catch (Exception except){
            except.printStackTrace();
        }
    }
    //Inserting Records in a Table
private static void insertStudents(int id, String studFName, String studLName){
try{
stmt = conn.createStatement();
stmt.execute("insert into " + tableName + " values (" +
id + ",'" + studFName + "','" + studLName +"')");
stmt.close();
        }
        catch (SQLException sqlExcept){
            sqlExcept.printStackTrace();
        }
    }
    //Displaying the values of the Table
    private static void selectStudents() {
        try {
            stmt = conn.createStatement();
ResultSet results = stmt.executeQuery("select * from " + tableName);
```

```
                ResultSetMetaData rsmd = results.getMetaData();
                int numberCols = rsmd.getColumnCount();
                for (int i=1; i<=numberCols; i++){
                    //print Column Names
                    System.out.print(rsmd.getColumnLabel(i)+"\t\t");
                }
    System.out.println("\n-----------------------------------------");
                while(results.next()){
    int id = results.getInt(1);
    String studFName = results.getString(2);
    String studLName = results.getString(3);
    System.out.println(id + "\t\t" + studFName + "\t\t" + studLName);
                }
                results.close();
                stmt.close();
            }
            catch (SQLException sqlExcept){
                sqlExcept.printStackTrace();
            }
        }
        //Closing the Connection
        private static void shutdown(){
            try{
                if (stmt != null){
                    stmt.close();
                }
                if (conn != null){
                    DriverManager.getConnection(dbURL + ";shutdown=true");
                    conn.close();
                }
            }
            catch (SQLException sqlExcept){}
        }
    }
```

In the preceding example, four methods are called into the `main()` method: The `createConnection()` method installs a Derby client driver and then creates a database `Student_DB`, where `scott` and `tiger` are the username and password of this database, respectively. The `insertStudents()` method takes three arguments, namely, `id`, `studFName`, and `studLName`, and inserts these arguments into the table `students`. The `selectStudents()` method is used to display the content of the table. The last method of the program, that is, shutdown(), closes all statements and shutdown connection properly.

You should create the Student_DB database and students table in Derby before running the given program.

Summary

In this chapter, you have learned about the Java DB, including features, history, architecture, and drivers of Java DB. Then the chapter has provided a comparison between Java DB and MySQL. It has also provided various Java DB product documentation, Derby tools, and utilities. You have learned the installation and configuration of Java DB on Windows operating system. Further, you have learned about the creation, deletion, and connection of Java DB database in embedded mode as well as network mode. In the section Java DB database encryption, you have learned how to encrypt/decrypt your database using a password. Finally, you have learned about using Java DB with Java application.

In the next chapter, you will learn about Java FX.



27

Java FX

If you need an immediate solution to:	*See page:*
Creating Simple JavaFX Application	981
Creating JavaFX Application on NetBeans IDE	981
Using the UI Controls	989
Using Visual Effects	989
Using Canvas API	990

In Depth

JavaFX is a collection of various advance features to strengthen the Java programming language. The inclusion of Scene Builder, Modena themes, FXML, 3DViewer graphics, etc., in JavaFX facilitates Java language in designing and developing rich interfaces with high-performance experiences for users. Generally, JavaFX is used in the form of an Application Programming Interface (API) that can directly be imported into an application. In today's age of the Internet, we find that the role of attractive user interfaces has increased. Any purchase, sale, or demonstration depends on how creative and well-designed interface has been provided to it. JavaFX can be used extensively to add to the attractiveness quotient of user interfaces. You can use the Scene Builder software package of JavaFX to create powerful designs without any requirements to write the code and script for it. Moreover, JavaFX applications are created using the standard Java language; hence, you need not worry about the consistency and availability. The most important part of developing JavaFX applications is the use of themes, Three-Dimensional (3D) viewers, Scene Graph APIs, graphics software and hardware, thread, pulse, media and images, etc. JavaFX provides Web component and User Interface (UI) controls in the layout to include some visual effects that are responsible for attracting the eyes of users.

In this chapter, you will learn about the general concept of JavaFX. Further, you will come to know about JavaFX features. You will learn about the availability of JavaFX for various platforms. In addition, you will get to know the scope of developing different kinds of applications using JavaFX APIs. Also, you will learn about JavaFX architecture and various components of it. Toward the end, you will come to know about the downloading and running of sample JavaFX applications and the use of Scene Builder in JavaFX.

Working with JavaFX

JavaFX is developed as an API in Java to help application developers in providing rich client applications. These applications can be designed, created, tested, debugged, and deployed by using JavaFX, which involves different types of graphics and media APIs. A JavaFX application delivers consistency in its performance across various platforms.

JavaFX Applications

The JavaFX library is included in the Java language in the form of an API. Therefore, application code for JavaFX has the capability of accessing any Java library API. As an instance, you can use Java libraries to access Java language APIs that support native system capabilities. Also, Java APIs can be used to connect to middleware applications that are based on the server.

With the help of Cascading Style Sheets (CSS), the look and feel of JavaFX can be customized so that you can differentiate style and appearance from your implementation. The designers can take care of all the required customizations in the appearance and style of JavaFX applications, and the developers are allowed to focus on the coding section only. People who are aware of the Web design practices or those who want the back-end logic and UI to be separated can use FXML scripting language for developing the UI presentation aspects and Java code for developing the application logic. Moreover, with the help of JavaFX Scene Builder, UIs can be developed without the need of writing the code. You can directly create the markup for your UI design by using Scene Builder. You can import the FXML markup into an Integrated Development Environment (IDE) in order to make it easy for the developer to add the business logic.

JavaFX Availability

JavaFX APIs are an integral feature of the Java Runtime Environment (JRE) and the Java Development Kit (JDK). These APIs are supported by applications that are compiled to JDK 7 or later on any of the leading Operating Systems (OSs), including Windows, Mac OS X, Linux, etc., because JDK is supported on all of these OSs. These APIs started supporting the Advanced RISC Machine (ARM) platform also with JavaFX 8. JDK for ARM platform involves a number of components, which include JavaFX base, graphics, and control components.

JavaFX applications are compatible across various OSs. Therefore, developers and users find a consistent runtime experience from these applications. Additionally, Oracle brings all its updates and new releases synchronously for all platforms. Also, companies in which mission-critical applications are being implemented to get the benefit of Oracle's support offered in the form of extensive support program. Therefore, you would never face compatibility or availability-related issues in creating JavaFX applications.

You can check the JDK download page to download sample applications of JavaFX in the form of a zip file that contains code and snippets for several JavaFX programs demonstrating the use of JavaFX to develop applications.

Key Features of JavaFX

JavaFX APIs are loaded with various features. Most of the features were introduced in JavaFX 8 and are its subsequent releases. Some of the primary features introduced in JavaFX 8 are as follows:

- **Java APIs—**JavaFX classes and interfaces are written in Java language code and provided in the Java library in the form of APIs. Some Java Virtual Machine (JVM) languages, such as JRuby and Scala, can be interchanged by these APIs.
- **FXML and Scene Builder—**FXML is an XML-based declarative markup language used to construct UI of a JavaFX application. An interactive Graphical User Interface (GUI) can be designed by a designer either through coding in FXML or through JavaFX Scene Builder. FXML markup generated in Scene Builder will be further ported to an Integrated Development Environment (IDE) for adding the business logic by the developer.
- **WebView—**It is a Web component that runs JavaScript code to enable embedding of Web pages within JavaFX applications through a technology called WebKitHTML. Java APIs can be accessed using the WebView JavaScript that can be called by Java APIs. JavaFX 8 involves support for various features of HTML5, which includes Web Sockets, Web Workers, Web Fonts, printing capabilities, etc.
- **Swing interoperability—**JavaFX provides features that can help you in updating your existing Swing applications. Some of the highly important features include rich graphics, media playback, and embedded Web content. JavaFX 8 involves an additional class named SwingNode, which you can use in your JavaFX applications to embed Swing content.
- **Built-in UI controls and Cascading Style Sheets (CSS)—**JavaFX has built-in features for controlling the UI and adjusting the CSS. These features help you in designing better, more attractive, and full-featured application UIs. You can develop the interface of various components in your application using CSS. Moreover, JavaFX 8 release involves DatePicker and TreeTableView UI controls that you can use in your application to provide more attractiveness and interactivity to the UI. Another update in JavaFX 8 release is the availability of CSS Styleable* classes in the form of public API to allow the styling of objects through CSS.
- **Modena theme—**JavaFX 8 has replaced the Caspian theme with the Modena theme as its default theme for developing JavaFX applications. However, you can still get and use the Caspian theme for JavaFX by adding the `setUserAgentStylesheet(STYLESHEET_CASPIAN)` line in the `start()` method of your application.
- **3D graphics features—**Java provides new API classes for LightBase (AmbientLight and PointLight subclasses), Shape3D (Box, Cylinder, MeshView, and Sphere subclasses), SceneAntialiasing, Material, SubScene, PickResult, etc., to the 3D graphics library in JavaFX 8. Also, this release provides an updated CameraAPI class.
- **Canvas API—**You can use the Canvas API to directly create drawing within an area of the JavaFX scene that comprises one graphical element (node).
- **Printing API—**JavaFX 8 includes an additional package, javafx.print, in the Java SE 8 release which enables public classes to access the JavaFX Printing API.
- **Rich-text support—**The JavaFX 8 release involves complex and extensive support for text scripts. It supports bi-directional text and complex scripts, including Thai and Hindi, in controls. Also, text nodes involve multiline, multi-style text to help developers create more attractive applications.

- **Multitouch support—**JavaFX can be used to design applications in which multitouch operations are performed. The multitouch support in JavaFX is based on the capabilities of the platform on which the application is developed.
- **Hi-DPI support—**The eighth release of JavaFX incorporates Hi-DPI display support in applications.
- **Hardware-accelerated graphics pipeline—**JavaFX package involves a pipeline (Prism) to render graphics-based application interfaces. Prism quickly provides smooth JavaFX graphics with the help of a supported graphics card or Graphics Processing Unit (GPU). Prism can also use software-rendering stack to provide smooth graphical experiences on encountering a GPU that is not supported by JavaFX.
- **High-performance media engine—**You can run multimedia content on the Web in the playback through JavaFX media pipeline support. The GStreamer multimedia framework offers a stable environment to JavaFX 8 with low latency.
- **Self-contained application deployment model—**JavaFX 8 includes self-contained application packages with which all of the application resources are provided. Additionally, a private copy of the Java and JavaFX runtimes are also available with these packages. The distribution of these packages can be installed as native applications, offering the same experience of installation and launching as native applications on that platform.

Application Development Scope of JavaFX

You can use JavaFX to build various types of applications that typically include cross-platform, network-aware applications. These applications offer GUI-based access to information by using high-performance modern interfaces that contain audio, video, graphics, animation, etc. JavaFX 8 includes several types of applications. The following list explores some of these applications:

- JavaFX Ensemble8
- Modena
- 3DViewer
- JavaFX Architecture
- Scene Graph in JavaFX
- Java Public APIs for JavaFX Features
- Graphics System
- Glass Windowing Toolkit
- Threads
- Pulse
- Media and Images
- Web Component
- CSS
- UI Controls
- Layout
- 2-D and 3-D Transformations
- Visual Effects

JavaFX Ensemble8

JavaFX provides a gallery of sample applications in the form of Ensemble8 to illustrate a large set of JavaFX features, such as animation, charts, and controls. These samples, which also involve their descriptions, can be run on all operating systems. We can view and interact with these samples, and also copy their source code on our desktop platform where the properties of various samples can be altered and adjusted as per the user's requirements and vision. Moreover, if the Internet facility is available, we can access API documentation, as relevant for us, through Ensemble8, which is designed to run on the ARM processor using JavaFX.

Modena

The look and feel of the UI components of JavaFX applications is demonstrated through Modena, which is a sample application that contains attractive themes. Moreover, you can compare the Modena theme with the traditional Caspian themes to explore and utilize various facets of both the themes.

Figure 27.1 shows an application screenshot developed in Modena theme:



Figure 27.1: Modena Theme on Desktop

3DViewer

You can use the 3DViewer sample application for navigation and examination of three-dimensional scenes by using a computer mouse or trackpad. Also, you can import a subset of Maya and Object (OBJ) files in JavaFX applications by using the 3DViewer importers. Moreover, animation files created in Maya can be imported through 3DViewer.

NOTE

While saving Maya files, you need to delete the construction history on all the objects to enable importing of the files.

Additionally, you can export your contents, too, in the form of Java files or FXML files by using 3DViewer. Figure 27.2 shows a screenshot of 3DViewer:

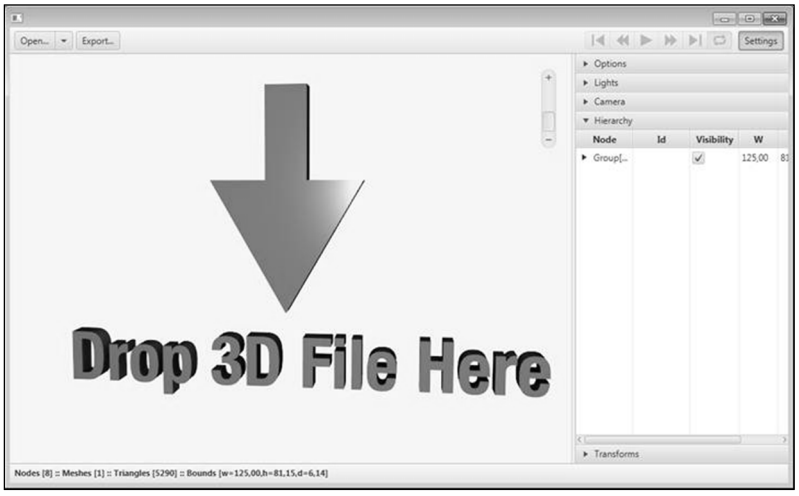


Figure 27.2: 3DViewer Application

The sample JavaFX applications can be downloaded from the official website of Oracle and run on your desktop system. However, you need to understand the JavaFX architecture so that you can develop JavaFX applications on your own.

JavaFX Architecture

JavaFX applications follow a specific architecture pattern in which various parts are interconnected. Figure 27.3 shows the different components of JavaFX Architecture:

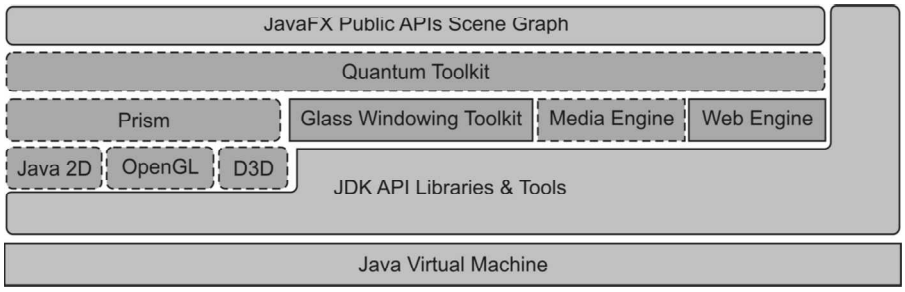


Figure 27.3: JavaFX Architecture Components

It can be seen in Figure 27.3 that the Quantum Toolkit in the JavaFX architecture lies beneath the JavaFX Public APIs and Scene Graph. Quantum Toolkit is used as an engine to run the JavaFX code. The subcomponents of this engine include JavaFX graphics processing engine (Prism), a small windowing system (Glass Window Toolkit) to provide efficient processing, a media engine, and a Web engine. All these components cannot be accessed publicly, but you need to know about these components to understand how applications created in JavaFX are run.

Let us learn about the different components of JavaFX architecture.

Scene Graph in JavaFX

The construction of JavaFX applications starts with scene graph, which is depicted as a part of the top layer in Figure 27.3. JavaFX scene graph involves a number of nodes in a hierarchical tree structure, wherein the visual elements in the UI of the application are reflected by each of the nodes. You can use JavaFX scene graph for input-handling purposes.

Node is the term given to a single element of a scene graph. You need to specify node ID, style class, and bounding volume to each node in JavaFX. Moreover, except for the root node of scene graph, all nodes have a parent and children (which may be zero or more); the root node has no parent. Additionally, nodes may be associated with the following aspects:

- Effects involving blurs and shadows
- Opacity
- Transforms
- Event handlers (including mouse, key, and input method)
- An application-specific state

Graphics primitives, which are not part of Java Swing and Abstract Window Toolkit (AWT), are provided with JavaFX scene graph. These elements are rectangles, text, controls, layout containers, images and media, etc.

Usually, scene graph is used for simplifying your work when dealing with UIs. Scene graph is particularly useful while working with rich UIs. For example, `javafx.animation` APIs and declarative methods, such as XML doc, enable you to quickly animate a number of graphics in the scene graph. You can create different types of content by using javafx.scene API. For instance, you can use the API to create the following elements:

- **Nodes—**Nodes can be shapes (2-D and 3-D), images, media, embedded Web browser, text, UI controls, charts, groups, containers, etc.
- **State—**State means transforms (positioning and orientation of nodes), visual effects, other visual state of the content, etc.
- **Effects—**Effects are simple objects that modify the appearance of scene graph nodes, such as blurs, shadows, and color adjustment.

Java Public APIs for JavaFX Features

JavaFX Public APIs are also part of the top layer itself in the JavaFX architecture, (Figure 27.3). This layer offers you to use a complete set of Java public APIs that can help you in developing rich client applications using JavaFX. All the freedom and flexibility required for the creation of rich client applications can be availed of through these APIs. The best capabilities of Java platform, such as comprehensive, immersive media functionality, are incorporated into the JavaFX platform that has been developed as an intuitive and comprehensive one-stop development environment. The features of Java APIs are as follows:

- The robust features of Java, such as multithreading, generics, annotations, and Lamda expressions, can be used.
- Java Web developers are allowed to use dynamic languages running on JVM, such as Groovy and JavaScript.
- Complex JavaFX applications are allowed to be developed by developers who can do so by using other system languages such as Groovy to write lengthy code
- Binding that supports high-performance lazy binding, binding expressions, bound sequence expressions, and partial bind reevaluation is permitted to be used. Moreover, you can use the binding library with alternative languages also for introducing the binding syntax. The binding syntax for alternate languages such as Groovy is similar to that of JavaFX Script.
- Observable lists and maps can be included by extending Java collections. Wiring UIs to data models, detecting alterations in these data models, and upgrading the corresponding UI control accordingly are allowed in your applications with the help of these maps and lists.

All the APIs and JavaFX programs are developed on the same model that is a continuation of the model introduced in the JavaFX products of version 1.x. Most of the JavaFX APIs are directly incorporated into Java. However, in the latest JavaFX release, some APIs that include Layout and Media in association with several other details have been improved and simplified on the basis of user feedbacks received for the JavaFX 1.x release. Generally, JavaFX is based on the Web standards. For example, the styling controls are defined on the basis of CSS standards and accessibility is specified on the basis of Accessible Rich Internet Applications (ARIA) standards.

Graphics System

Below the top layer in the architecture (Figure 27.3) is the JavaFX graphics system, which is a detail of the implementation below the JavaFX scene graph layer. Graphics system provides support for both 2D and 3D scene graphs. Moreover, you can render graphics through the graphics software system if in case hardware-accelerated rendering is inadequate for supporting by the graphics hardware on your system.

JavaFX platform uses the following two graphics-accelerated pipelines:

- **Prism—**These processes are used for rendering jobs. You can use them on both hardware and software renderers, which also include 3D graphics. Moreover, prism processes are responsible for rasterizing and rendering JavaFX scenes. The device being used by you determines the following multiple render paths:
  - DirectX 9 on Windows XP and Windows Vista
  - DirectX 11 on Windows 7
  - OpenGL on Mac, Linux, Embedded
  - Software rendering when hardware acceleration is not possible

  Moreover, whenever possible, you can use paths that are fully accelerated by hardware. However, in case this possibility does not exist, we use the software render path to exploit the already distributed software render path in the Java Runtime Environments (JREs). This is especially significant in handling 3D scenes, but the performance of hardware rendering paths is better in comparison with that of the software rendering paths.

- **Quantum Toolkit—**Quantum Toolkit is the other pipeline used in JavaFX platform through which Prism and Glass Windowing Toolkit associate together to make them selver available to the JavaFX layer. In addition, the management of threading rules associated with rendering versus events handling is done by the threading rules and the Quantum Toolkit.

Glass Windowing Toolkit

In Figure 27.3, you can see that in the JavaFX graphics stack, the Glass Windowing Toolkit is at the lowest level. The main responsibility of the Glass Windowing Toolkit is to provide native operating services, which includes managing the windows, timers, and surfaces. It acts as the platform-dependent layer used for connecting the JavaFX platform with the native operating system.

Moreover, the Glass Windowing Toolkit also manages the event queue. It schedules the usage of thread by using the native operating system's event queue functionality. The Glass Windowing Toolkit executes on the same thread on which the JavaFX application executes.

Threads

JavaFX involves different kinds of threads to support proper running of applications. A system can run two or more of the following threads at any instant:

- **JavaFX application thread—**Refers to the primary thread used by the developers of JavaFX application. A "live" scene, which is a scene that is part of a window, can be accessed through this thread. In a background thread, manipulation and construction of a scene graph can be done. However, accessing of a scene graph from the JavaFX application thread must be permitted if the scene graph's root node is bound to the live object of any scene. Therefore, developers can build complex scene graphs on a background thread by keeping smooth and fast animations on 'live' scenes. You must be careful while embedding JavaFX code into Swing applications because the JavaFX application thread is not similar to thread from the Swing and AWT Event Dispatch Thread (EDT).
- **Prism render thread—**Refers to the thread that can handle rendering independently from the event dispatcher. It enables rendering of frame N while frame N+1 is in processing. Therefore, the concurrent processing ability provided by this thread is a big advantage for modern systems having multiple processors. The Prism render thread handles off-load work in rendering with the help of multiple rasterization threads.
- **Media thread—**This thread can execute in the background for synchronizing the latest frames through the scene graph with the help of JavaFX application thread.

Pulse

A pulse can be referred to as an event that directs the JavaFX scene graph about the time of synchronizing the state of the elements on the scene graph with Prism. A pulse can be regulated at maximum of 60 frames per second (fps) and is fired whenever animations are being executed on the scene graph. A pulse can be scheduled even when an animation is not running and something is modified in the scene graph. For instance, a pulse can be scheduled when the position of a label control is changed.

Application developers can use a pulse for handling events asynchronously, enabling the system for batch execution of events on the pulse.

Layout and CSS are also attached to pulse events. When numerous changes are done in the scene graph, it might lead to multiple updates to layout or CSS, which might degrade the pulse performance to a great extent. The degradation in performance of the system is managed automatically by it. The system performs a CSS and layout pass once per pulse for avoiding performance degradation. The passes can be triggered manually by application developers as required for taking measurements prior to a pulse.

The pulse events are executed by the Glass Windowing Toolkit, which utilizes the high-resolution native timers for the execution.

Media and Images

The javafx.scene.media APIs provide the JavaFX media functionality which is provided as three separate components: the Media object represents a media file, the MediaPlayer plays the media file, and a MediaView is a node that displays the media. Both visual and audio media are supported by JavaFX. The MP3, Audio Interchange File Format (AIFF), and Waveform Audio File (WAV) format of audio files, and Flash Video (FLV) format of video files are supported. The Media Engine component is created for providing performance and stability and enabling steady behavior across platforms.

Web Component

The Web component is a JavaFX UI control based on Webkit, which is an open-source Web browser engine in which HTML5, CSS, JavaScript, Document Object Model (DOM), and Scalable Vector Graphics (SVG) are supported. This component uses its API for providing a Web viewer and full browsing functionality. The Web component allows developers for implementing the following features in their Java applications:

- Rendering to HTML content from local or remote URL
- Support to history, and Back and Forward navigation
- Reloading the content
- Applying effects to the Web component
- Editing of the HTML content
- Execution of the JavaScript commands
- Handling of events

This embedded browser component comprises the following classes:

- **WebEngine—**It provides the capability of browsing the Web page.
- **WebView—**It encapsulates a WebEngine object that embeds HTML content into an application's scene. It provides fields and methods for applying effects and transformations. This class is an extension of the Node class.

CSS

JavaFX CSS allows customized styling to the JavaFX application's UI without modifying the source code of application. You can apply CSS to any JavaFX scene graph's nodes asynchronously. JavaFX CSS styles can be applied to the scene at runtime, thus allowing dynamic change to the appearance of an application. Figure 27.4 displays the implementation of two different CSS styles to the same set of UI controls:

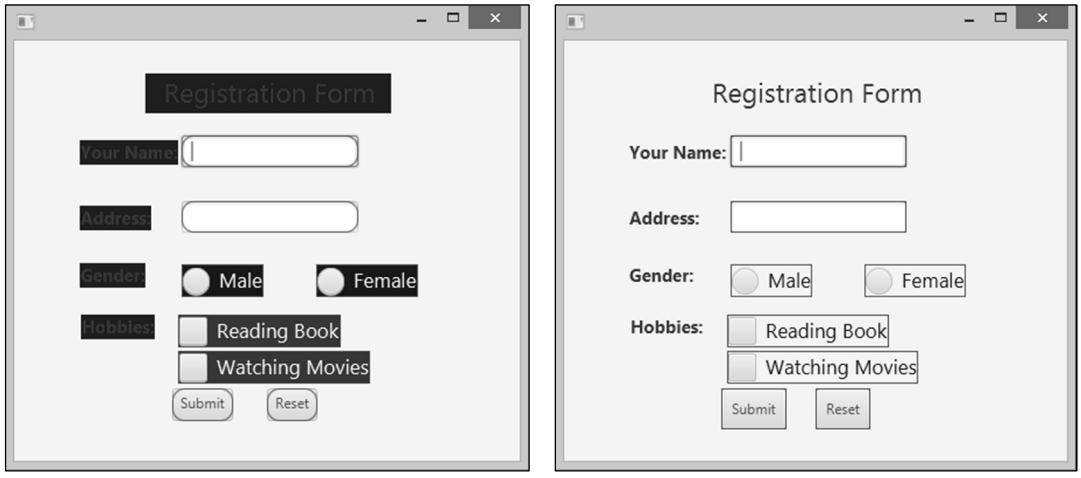


Figure 27.4: CSS Style Sheet Sample

UI Controls

JavaFX API provides the JavaFX UI controls that are created by using nodes in the scene graph. These controls utilize full benefit of the visually rich features of the JavaFX platform and provide portability across various platforms. The theme and skins of the UI controls can be set using the JavaFX CSS. JavaFX UI control components involve the following elements:

- Label
- Button
- Radio Button
- Toggle Button
- Checkbox
- Choice Box
- Text Field
- Password Field
- Scroll Bar
- Scroll Pane
- List View
- Table View
- Tree View
- Tree Table View
- Combo Box
- Separator
- Slider
- Progress Bar and Progress Indicator
- Hyperlink
- Tooltip
- HTML Editor
- Titled Pane and Accordion
- Menu
- Color Picker
- Date Picker
- Pagination Control
- File Chooser
- Customization of UI Controls
- UI Controls on Embedded Platforms

Layout

Within a scene graph of a JavaFX application, the arrangements of the UI controls can be made flexible and dynamic with the help of layout containers or panes. The JavaFX Layout API provides the following container classes that allow automation of common layout models:

- The `BorderPane` class structures its content nodes in the top, bottom, right, left, or center region
- The `HBox` class structures its content nodes horizontally in a single row.
- The `VBox` class structures its content nodes vertically in a single column.
- The `StackPane` class keeps its content nodes in a back-to-front single stack.
- The `GridPane` class allows the creation of a flexible grid of rows and columns.

- The `FlowPane` class structures its content nodes in either a horizontal or vertical flow.
- The `TilePane` class structures its content nodes in evenly sized layout cells or tiles.
- The `AnchorPane` class allows developers to build anchor nodes to the top, bottom, left side, or center of the layout

Different containers can be nested within a JavaFX application in order to attain a desired layout structure.

2-D and 3-D Transformations

You can transform each node in the JavaFX scene graph to the x-y coordinate by using the following javafx.scene.tranform classes:

- **translate—**Allows transferring of a node from one location to another along the x, y, and z planes relative to its initial position.
- **scale—**Allows resizing of a node to seem either larger or smaller on the x, y, and z planes on the basis of the scaling factor.
- **shear—**Allows rotation of one axis so that the x-axis and y-axis do not remain perpendicular. The node's coordinates are then shifted by the mentioned multipliers.
- **rotate—**Allows rotation of a node about a specific pivot point of the scene.
- **affine—**Performs a linear mapping from 2-D/3-D coordinates with other 2-D/3-D coordinates while preserving the 'straight' and 'parallel' properties of the lines. You must note that this class should be used with Translate, Scale, Rotate, or Shear transform classes instead of being used directly.

Visual Effects

Visual Effects are used for the creation of rich client interfaces in the JavaFX scene graph for enhancing the appearance of JavaFX applications in real time. These JavaFX Effects are primarily image-pixel-based; and, therefore, they use the group of nodes present in the scene graph, render it as an image, and then apply the specific effects to it. The following effects are available in Visual Effects in JavaFX:

- Blend Effect
- Bloom Effect
- Blur Effects
- Drop Shadow Effect
- Inner Shadow Effect
- Reflection
- Lighting Effect
- Perspective Effect

Visual Effects in JavaFX use the following classes:

- **Drop Shadow—**Renders a shadow of a provided content behind the content
- **Reflection—**Renders a reflected version of the content below the actual content
- **Lighting—**Simulates a light source shining on a given content and provides a realistic three-dimensional appearance to a flat object

Introducing Scene Builder in JavaFX

A visual layout environment is provided by JavaFX Scene Builder which allows quick designing of UI for JavaFX applications without the need of writing code. It enables simple drag-and-drop positioning of GUI components onto a JavaFX scene. After building the layout for your UI, the FXML code for the layout gets generated automatically. JavaFX Scene Builder provides simple yet spontaneous interface that can even help non-programmers to swiftly prototype interactive applications that connect GUI components to the application logic.

Intended Audience

JavaFX applications are of interest for Web designers and developers who can apply their creativity and learning toward improving the experience of end users. Hence, the target audience for JavaFX Scene Builder is as follows:

- **Java developers—**Refer to persons who can rapidly prototype the client application's GUI layout and create the application logic independently.
- **Designers—**Refer to persons who can rapidly prototype the client application's GUI layout without the need of any application code to be written first. Moreover, they can design and preview the GUI layout and define its appearance using style sheets.

Key Features of JavaFX Scene Builder

JavaFX Scene Builder provides several features that can help you to design attractive user interfaces so that the user experience could be improved. Some of the interesting features of JavaFX Scene Builder include the following:

- **A drag-and-drop WYSIWYG interface** enables you to rapidly develop a GUI layout without the need of writing source code. You can use the library of GUI controls and the content panel for adding, combining, and editing JavaFX GUI controls in your layout.
- **Tight integration with the NetBeans IDE** enables optimum development workflow.
- **Easy integration with any Java IDE** because it is a standalone development tool.
- **Automatic FXML code generation** takes place when you create and alter the layout of your GUI. The FXML code gets stored in a separate file from the source of application logic and style sheet files.
- **Live editing and preview features** allow you to rapidly visualize the GUI layout modifications that you have done without any need of compilation. These features help in minimizing the development time for your application. You can also apply CSS to your GUI layout and preview the created look and feel.
- **Accessibility to the full JavaFX GUI controls library** is provided. To see the full list of supported JavaFX 8 GUI components, type FX8 in the Library panel's Search text field. The list includes the TreeTableView, DatePicker, and SwingNode components.
- **Ability to add custom GUI components to the Library** is now provided. The Library of available GUI components can be extended by importing customized GUI components from third-party JAR files, FXML files, or adding them from the Hierarchy or Content panels.
- **3D support** is provided. FXML documents containing 3D objects can now be loaded and saved in the Scene Builder 2.0 tool. You can view and edit properties of the 3D objects using the Inspector panel (Material and Mesh complex properties are not yet supported). You cannot, however, create new 3D objects using the Scene Builder tool.
- **Support for rich text** has been added. A new container TextFlow is now available in the Library of GUI components. You can drag multiple text nodes and other types of nodes into the TextFlow container. You can also directly manipulate the text nodes to re-arrange them in the container. Inline and property editing features are also available for each text node.
- **JavaFX Scene Builder Kit** is provided with Scene Builder 2.0. It is an API that enables the incorporation of Scene Builder panels and functionalities into the GUI of a larger application or a Java IDE, such as NetBeans, IntelliJ, and Eclipse.
- **CSS support** allows flexible management of the appearance of your application's UI.
- **Cross-platform support** is provided for Windows, Linux, and Mac OS X operating systems.

Now that we have overviewed what's in this chapter, it is time to turn to the "Immediate Solutions" section.

Immediate Solutions

Creating Simple JavaFX Application

Notepad or any Integrated Development Environment (IDE) supporting the Java language (such as NetBeans, Eclipse, or IntelliJ IDEA) can be used for generating JavaFX applications, which are developed using the Java language.

Here, we are creating a simple JavaFX application using the Notepad editor:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.stage.Stage;

public class HelloJavaFX extends Application {
   public static void main(String[] args) {
    launch(args);
  }
  @Override
  public void start(Stage s) {
    Label lbl= new Label();
    lbl.setText("Hello JavaFX 8");
    Scene scn =new Scene(lbl);
    s.setTitle("Hello JavaFX");
    s.setScene(scn);
    s.setHeight(300);
    s.setWidth(300);
    s.show();
  }
}
```

Save the preceding code with the name HelloJavaFX.java. We can compile and run the JavaFX programs in the same way as we compile and run simple Java programs. To compile and run the preceding program, use the following commands on the command prompt:

```
C:\>javac  HelloJavaFX.java    //To compile the program
C:\>java HelloJavaFX       // To execute the compiled program
```

Here is the output of the preceding program, as shown in Figure 27.5:



Figure 27.5: Output of the HelloJavaFX.java Program

Creating JavaFX Application on NetBeans IDE

Before creating JavaFX applications on IDE, you must first install the following software on your computer:

- JDK 1.8
- NetBeans 7.5 or later
- JavaFX Scene Builder 2.0

After the successful installation of JDK 1.8 and NetBeans IDE, you need to install the JavaFX Scene Builder 2.0 by performing the following steps:

1. Download the JavaFX Scene Builder 2.0 setup from the **http://www.oracle.com/technetwork/java/javafxscenebuilder-1x-archive-2199384.html** link. Save the downloaded setup at an appropriate location in your computer.
2. After downloading, `double-click` the setup file to start the installation. The `JavaFX Scene Builder - 2.0 Setup` dialog box appears (Figure 27.6).
3. *Click* the `Next` button, as shown in Figure 27.6:

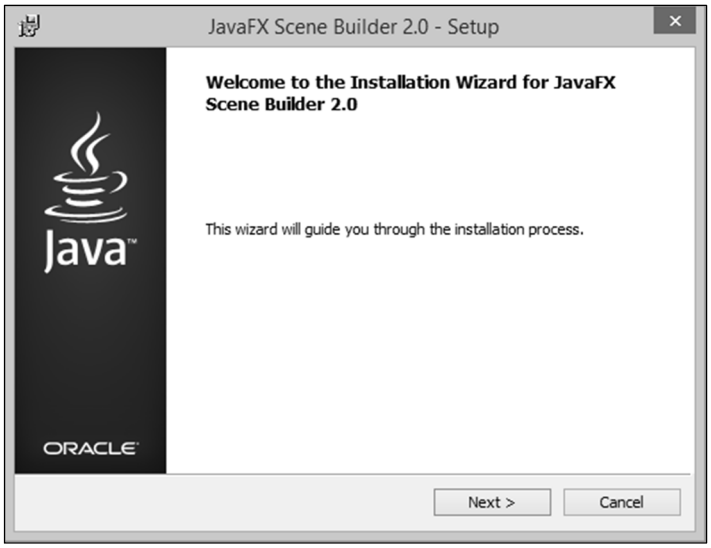


Figure 27.6: The JavaFX Scene Builder 2.0 - Setup Dialog Box

The `JavaFX Scene Builder 2.0 - Destination Folder` dialog box appears (Figure 27.7).

4. *Click* the `Install` button, as shown in Figure 27.7:



Figure 27.7: The JavaFX Scene Builder 2.0 – Destination Folder Dialog Box

The `JavaFX Scene Builder 2.0 - Progress` dialog box appears, as shown in Figure 27.8:

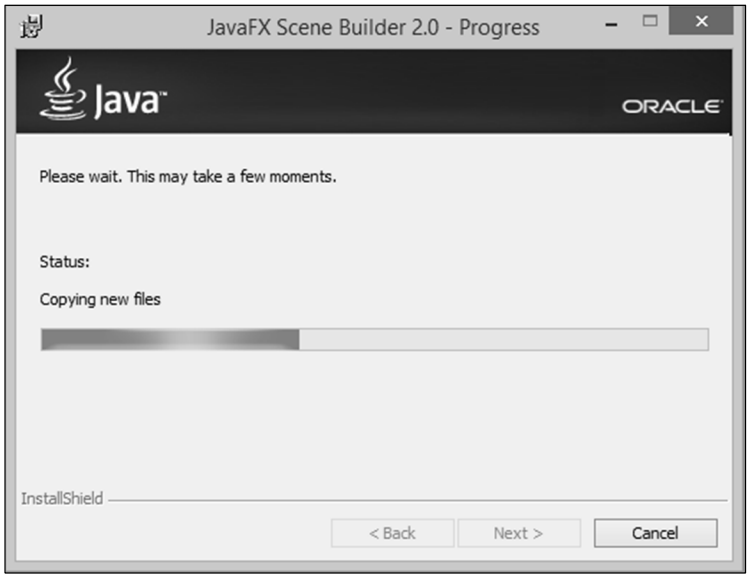


Figure 27.8: The JavaFX Scene Builder 2.0 – Progress Dialog Box

After completing the installation of JavaFX Scene Builder 2.0, the `JavaFX Scene Builder 2.0 - Progress` dialog box disappears. Now, the `JavaFX Scene Builder 2.0 - Complete` dialog box appears, as shown in Figure 27.9:

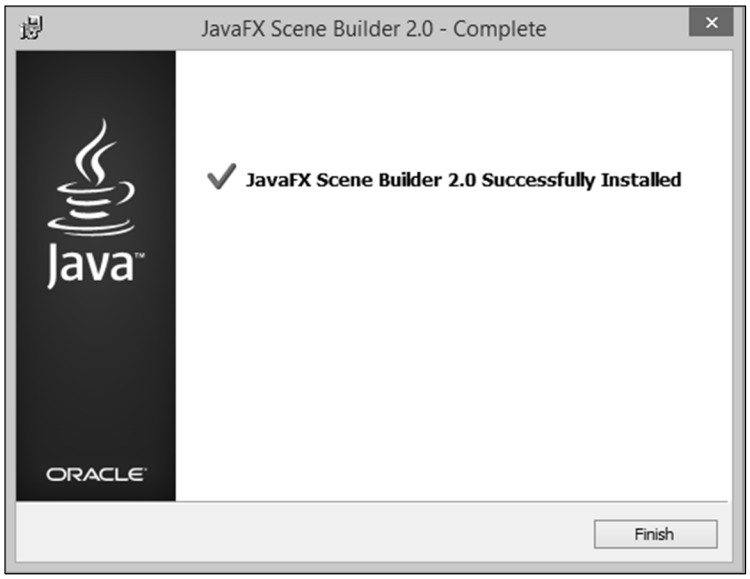


Figure 27.9: The JavaFX Scene Builder 2.0 – Complete Dialog Box

After successful installation of JavaFX Scene Builder 2.0, now perform the following steps to create a simple application in NetBeans 8 IDE:

1. *Open* the NetBeans IDE.

2. *Click* the `File`→`New Project` option in the `File` menu. The `New Project` dialog box appears (Figure 27.10).
3. `Select` the `JavaFX` folder in the `Categories` pane and then *select* the `JavaFX FXML Application` template in the `Projects` pane (Figure 27.10).

5. *Click* the `Next` button, as shown in Figure 27.10:

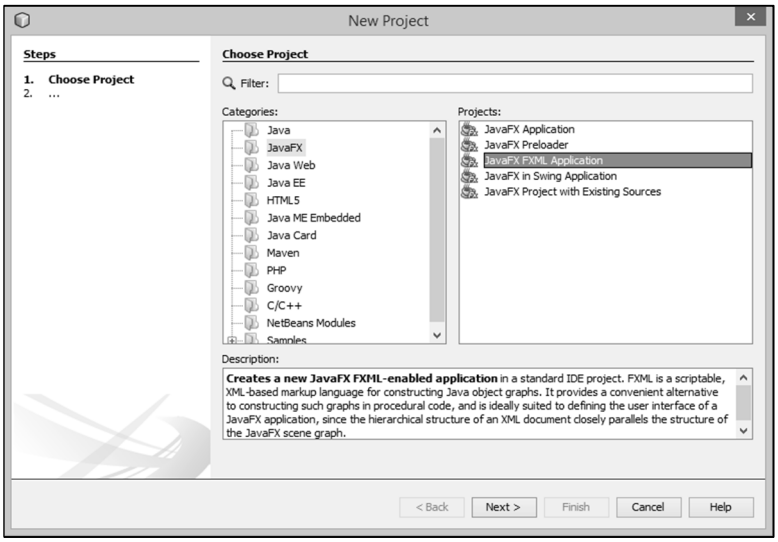


Figure 27.10: The New Project Dialog Box

The `New JavaFX Application` dialog box appears (Figure 27.11).

6. *Enter* the project name in the `Project Name` text box. In our case, we have entered `JavaFXApplicationDemo` (Figure 27.11).

NOTE

You can also set the location of the project according to your requirement.

7. *Click* the `Finish` button, as shown in Figure 27.11:

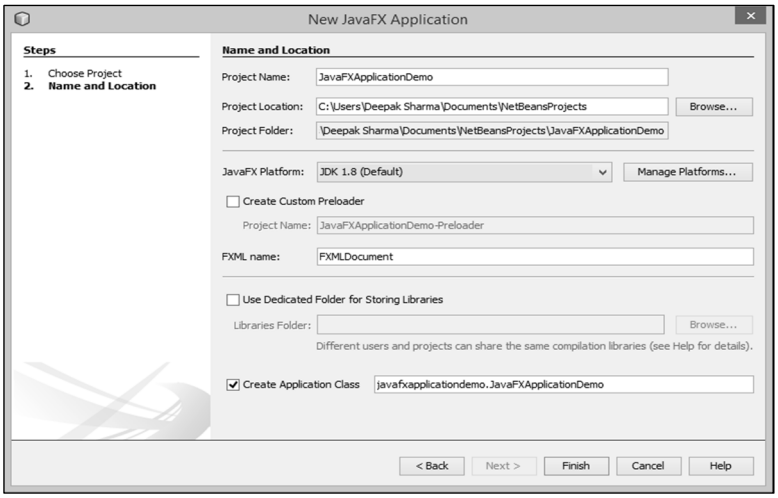


Figure 27.11: The JavaFX Application Dialog Box

8. The JavaFXApplicationDemo application is created and the `JavaFXApplicationDemo.java` file opens in NetBeans IDE, as shown in Figure 27.12:

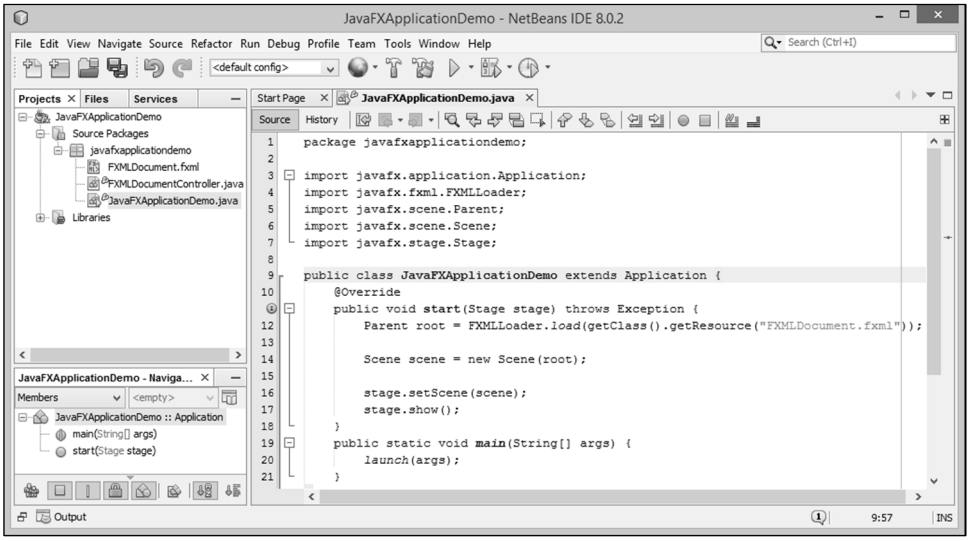


Figure 27.12: The JavaFXApplicationDemo Application

9. *Right-click* the `FXMLDocument.fxml` file in the `Projects` pane and `select` the `Open` option from the context menu, as shown in Figure 27.13:

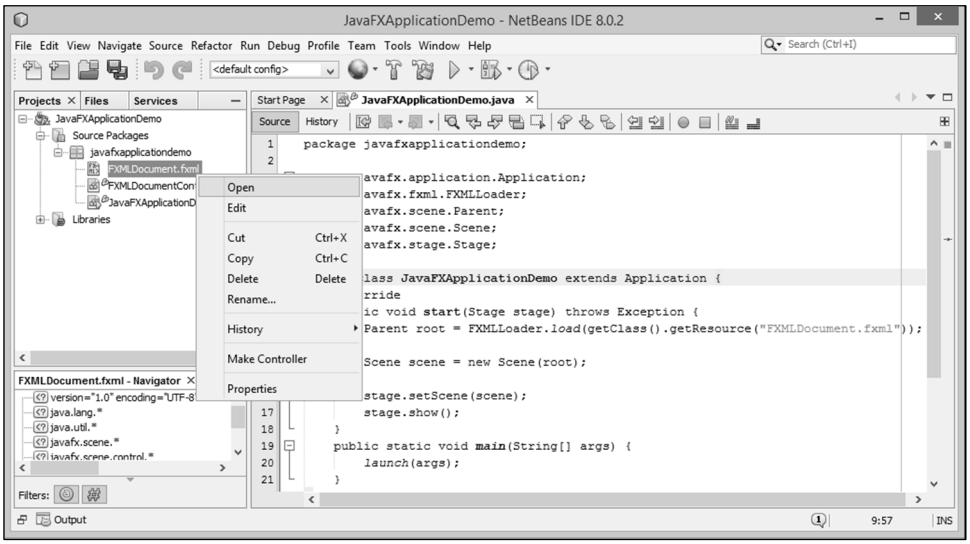


Figure 27.13: The JavaFXApplicationDemo Application

The FXMLDocument.fxml file opens in the JavaFX Scene Builder (Figure 27.14). Now, drag and drop some controls from the Controls section in the Library panel on the AnchorPane window. In our case, we have designed a Login window using four controls: such as Label, TextField, PasswordField, and Button. We have also set the Text property of the Label and Button controls in the Inspector panel, as shown in Figure 27.14:



Figure 27.14: The FXMLDocument.fxml Window

10. *Click* the *File* menu and then *select* the `Save` option from the context menu to save the file (Figure 27.14).

    Now, open the FXMLDocument.fxml file in NetBeans IDE and you can see that the following code is generated automatically by the NetBeans IDE:

```
 <?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.text.*?>
<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<AnchorPane id="AnchorPane" prefHeight="238.0" prefWidth="322.0"
xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="javafxapplicationdemo.FXMLDocumentController">
    <children>
        <Button fx:id="button1" layoutX="144.0" layoutY="161.0" onAction="#handle"
text="Submit" />
      <TextField fx:id="txtBox1" layoutX="144.0" layoutY="88.0" />
      <Label layoutX="14.0" layoutY="92.0" text="Enter Your Name:" />
      <Label fx:id="label2" layoutX="14.0" layoutY="195.0" prefHeight="25.0"
 prefWidth="286.0" />
      <Label layoutX="14.0" layoutY="126.0" text="Enter Your Password:" />
      <PasswordField fx:id="txtPass" layoutX="144.0" layoutY="122.0" />
      <Label layoutX="111.0" layoutY="24.0" text="Login Here">
         <font>
            <Font size="25.0" />
         </font>
      </Label>
    </children>
</AnchorPane>
```

11. *Right-click* the `FXMLDocument.fxml` file in the `Projects` pane and `select` the `Make Controller` option from the context menu, as shown in Figure 27.15:

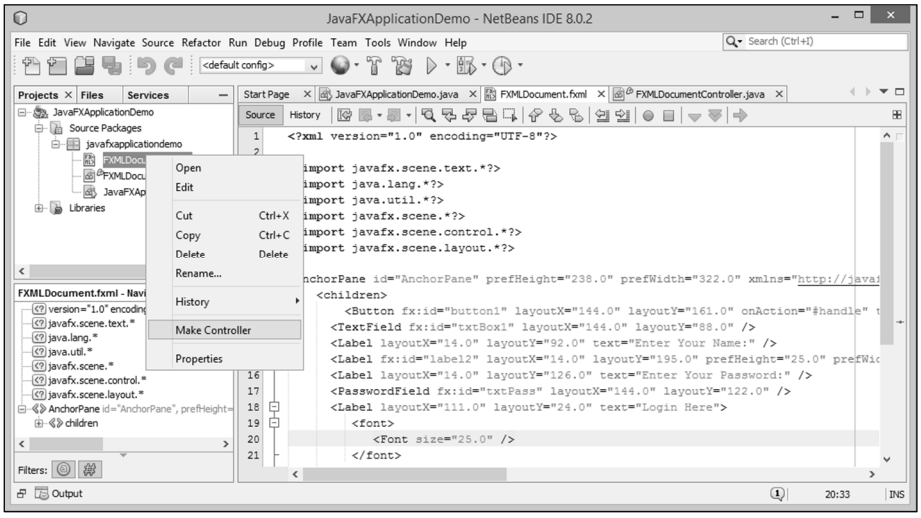


Figure 27.15: Using the Make Controller Option

After selecting the Make Controller option, the FXMLDocumentController.java file gets updated and the highlighted code is generated automatically in it, as shown in the following code:

```
package javafxapplicationdemo;
import java.net.URL;
import java.util.ResourceBundle;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.*;

public class FXMLDocumentController implements Initializable {
    @FXML
    private Label label2;
    @FXML
    private Button button1;
    @FXML
    private TextField txtBox1;
    @FXML
    private PasswordField txtPass;
    @Override
    public void initialize(URL url, ResourceBundle rb) {
    }
    @FXML
    private void handle(ActionEvent event) {

    }
}
```

12. *Import* the following package in the `FXMLDocumentController.java` file:

```
import javafx.scene.paint.Color;
```

13. *Add* the following code in the handle event of the Button control:

```
if(txtBox1.getText().equals("deepak") && txtPass.getText().equals("dreamtech")){
  label2.setTextFill(Color.BLACK);
  label2.setText("Welcome "+txtBox1.getText());
}
else{
  label2.setText("Invalid username or password!");
  label2.setTextFill(Color.rgb(210, 39, 30));
}
```

14. *Press* the `F6` key to run the JavaFXApplicationDemo application. The output appears, as shown in Figure 27.16:



Figure 27.16: Output of the JavaFXApplicationDemo Application

15. *Enter* username and password. In our case, we have set the username as "deepak" and password as "dreamtech". If you enter the correct username and password, then the message "Welcome deepak" gets displayed, as shown in Figure 27.17:



Figure 27.17: Output of the JavaFXApplicationDemo Application

Now, if you have entered incorrect username or password, then the message "Invalid username or password!" appears, as shown in Figure 27.18:



Figure 27.18: Output of the JavaFXApplicationDemo Application on Invalid Input

Using the UI Controls

There are various types of controls in the JavaFX API about which you have learned earlier. UI controls are usually used for creating visually rich and highly interactive application. Let's now learn how to use some of the useful UI controls in an application. In this application, we are using the ImageView class to add an image in the JavaFX application by using the following code:

```
import javafx.scene.image.*;
import javafx.stage.Stage;

public class AddingImage extends Application {

    @Override
    public void start(Stage s) {
        ImageView iV = new ImageView();
        iV.setImage(new Image("JavaFX.png"));
        StackPane sPane=new StackPane();
        sPane.getChildren().add(iV);
        Scene scene = new Scene(sPane);
        s.setTitle("Displaying Image");
        s.setHeight(400);
        s.setWidth(400);
        s.setScene(scene);
        s.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Save the preceding code with the name AddingImage.java. Use the following commands on the command prompt to compile and run the application:

```
C:\>javac AddingImage.java
C:\>java AddingImage
```

NOTE

In the preceding code, we have used an image with the name "JavaFX.png". This image should be present at the location where you have saved the .java file.

Here is the output of the preceding code, as shown in Figure 27.19:



Figure 27.19: Output of the AddingImage Application

Using Visual Effects

There are various kinds of visual effects in JavaFX which you have learned earlier. Now, we are creating a simple application in which we are applying 3D visual effect in a Scene by using the `PerspectiveCamera` class in the following code:

```
import javafx.application.Application;
```

```
import javafx.scene.*;
import javafx.scene.paint.*;
import javafx.scene.shape.*;
import javafx.scene.transform.*;
import javafx.stage.Stage;

public class Creating3DShapes extends Application {
    public static void main(String[] args) {
        launch(args);
    }
    @Override
    public void start(Stage stage) throws Exception {
        stage.setResizable(false);
        Scene scene = new Scene(content());
        stage.setScene(scene);
        stage.show();
    }
    public Parent content() throws Exception {
        Box box = new Box(8, 8, 8);
        PhongMaterial pMat=new PhongMaterial(Color.GOLDENROD);
        box.setMaterial(pMat);
        box.setDrawMode(DrawMode.FILL);
        PerspectiveCamera pCam = new PerspectiveCamera(true);
        pCam.getTransforms().addAll (
                new Rotate(-30, Rotate.X_AXIS),
                new Rotate(-30, Rotate.Y_AXIS),
                new Translate(0, 1, -20));
        Group g = new Group();
        g.getChildren().add(pCam);
        g.getChildren().add(box);
        SubScene scene = new SubScene(g, 600,550);
        scene.setFill(Color.BLUE);
        scene.setCamera(pCam);
        Group g1 = new Group();
        g1.getChildren().add(scene);
        return g1;
    }
}
```

Save the preceding code with the name "Creating3Dshapes.java". Use the following commands on the command prompt to compile and run the application:

```
C:\>javac Creating3DShapes.java
C:\>java Creating3DShapes
```

Here is the output of the preceding code, as shown in Figure **27.**20:

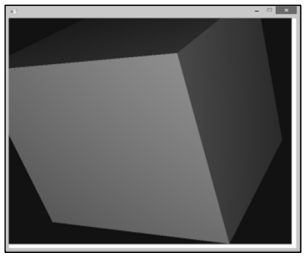


Figure 27.20: Output of the Creating3DShapes Application

Using Canvas API

You have learned earlier that we can use the Canvas API to directly create drawing within an area of the JavaFX scene that consists of one graphical element (node). Let's now create an application by using the Canvas API as follows:

```
import javafx.application.*;
```

```
import javafx.scene.*;
import javafx.scene.canvas.*;
import javafx.stage.Stage;

public class CanvasDemo extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage s) {
        Canvas cnvs = new Canvas();
        cnvs.setHeight(400);
        cnvs.setWidth(400);
        Group g = new Group();
        GraphicsContext gContext = cnvs.getGraphicsContext2D();
        drawShapes(gContext);
        g.getChildren().add(cnvs);
        Scene scn=new Scene(g);
        s.setTitle("Canvas Demo");
        s.setScene(scn);
        s.show();
    }

    private void drawShapes(GraphicsContext gContext) {
        gContext.setLineWidth(8);
        gContext.fillOval(10, 40, 20, 10);
        gContext.strokeOval(40, 50, 20, 20);
        gContext.fillRoundRect(100, 50, 20, 20, 5, 5);
        gContext.strokeRoundRect(140, 50, 20, 20, 5, 5);

        gContext.strokeLine(50, 20, 20, 50);

    }
}
```

Save the preceding code with the name "CanvasDemo.java". Use the following commands on the command prompt to compile and run the application:

```
C:\>javac CanvasDemo.java
C:\>java CanvasDemo
```

Here is the output of the preceding code, as shown in Figure 27.21:

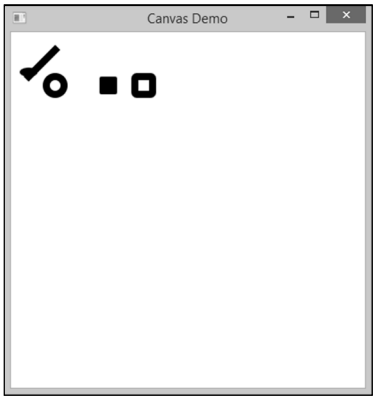


Figure 27.21: Output of the CanvasDemo Application

Summary

In this chapter, you have learned about the basic concept of JavaFX and its features. Next, you have learned about the availability of JavaFX for different platforms. In addition, you have studied the JavaFX architecture along with its different components. The chapter has also explained the development scope of various applications using JavaFX APIs. Furthermore, you have studied the process of downloading and running sample JavaFX applications. Finally, you have learned the use of JavaFX Scenen Builder software package.



Glossary

ABORT

It indicates that an image that was being tracked is aborted before its production was completed.

ALLBITS

It indicates that a static image that was previously drawn is now complete and can be redrawn in its final form.

Applet

An applet is a program written in the Java programming language that can be included in an HTML page in the same way as an image is included in a page.

Appletviewer

Facilitates to run Java applets.

Block

A block indicates that a thread is waiting for the lock or permit to access the resource.

Buffers

The Buffer class defines the core functionality—current position, limit, and capacity.

Charsets

A charset defines the way the bytes are mapped to characters.

Class

This is a template from which you can create objects. The definition of a class includes the formal specifications for the class and any data and methods in it.

Collection

It is the top of the collection hierarchy. It supports basic grouping of elements.

Command

Command consists of a list of strings which make the external program file to be invoked and its arguments, if any, significant.

Data members

These are variables that are part of a class. You use them to store the data the object uses. Objects support both instance variables, whose values are specific to the object, and class variables, whose values are shared among the objects of that class.

Deployers

The Deployers are responsible for deploying EJB applications and Web applications into the server environment.

Deque

It extends Queue to handle a double-ended queue.

Distributed

Java is a distributed language as it can be used to create applications to communicate over the network.

Dynamic

During runtime of a Java program, the relevant information that is used to verify and resolve access to objects is required. This concept of providing runtime information is referred to as dynamically linking the code.

Environment

A mapping, which is system-dependent varying from variables to values. The copy of the environment of current process is the initial value.

ERROR

It indicates that an image that was being tracked has encountered an error.

FRAMEBITS

It indicates that another complete frame of a multiframe image (which was previously drawn) is now available to be redrawn.

HEIGHT

It indicates that the height of the base image is now available (and can be read from the height argument of the imageUpdate method).

Inheritance

This is the process of deriving one class, called the derived class, from another, called the base class, and being able to make use of the members of the base class in the derived class.

JApplet

It is an extended version of java.applet.Applet that adds support for root panes and other panes.

jar

Serves as archive used to package related class libraries into a single executable JAR file and also helps to manage the JAR files.

java.applet

Provides classes that are necessary for creating an applet and those which are used to communicate with its applet context.

java.awt

Provides classes for creating user interface and painting graphics and images.

java.io

Provides classes for system input and output through data streams, serialization, and the file system.

java.lang

Provides classes that are fundamental to the design of the Java programming language.

java.net

Provides classes that are used for implementing networking in Java programs.

java.util

Provides legacy collection classes, event model, collection framework, date and time capabilities, internationalization, and other utility classes, such as string tokenizer.

Java

Serves as a Java interpreter used to run Java applets and applications by reading and interpreting bytecode files.

javac

Serves as a Java compiler used to translate Java source code to bytecode files.

javacdoc

Creates HyperText Markup Language (HTML) documentation for Java source code files.

javah

Serves as the C header and stub generator, which is used to write native methods.

javap

Serves as a Java disassemble used to convert bytecode files into a Java program description.

JButton

It is a push or command button.

JCheckBox

It is a checkbox that can be selected or deselected, displaying its state visually.

JCheckBoxMenuItem

It is a menu item that can be selected or deselected and displays its state visually.

JColorChooser

It is a pane of controls to allow a user to select a color.

JComboBox

It is a combo box, which is a combination of a text field and dropdown list.

JComponent

It is the base class for Swing components.

jdb

Serves as a Java debugger used to find errors in Java programs.

JDesktopPane

It is a container used to create a multiple-document interface or a desktop.

JDialog

It is the base class for creating a dialog window.

JEditorPane

It is a text component that allows the user to edit various kinds of content.

JFileChooser

It lets the user choose a file.

JFrame

It is an extended version of java.awt.Frame that adds support root panes and other panes.

JInternalFrame.JDesktopIcon

It represents an iconified version of a JInternalFrame component.

JInternalFrame
: It is a lightweight object that provides many of the features of a heavyweight frame.

JLabel
: It is a display area for a short text string or an image (or both).

JLayeredPane
: It adds layers to a Swing container, allowing components to overlap each other.

JList
: It is a component that allows the user to select one or more objects from a list.

JMenu
: It is a pop-up menu containing JMenuItem objects that's displayed when the user selects it in the JMenuBar component.

JMenuBar
: It is an implementation of a menu bar.

JMenuItem
: It is an implementation of a menu item.

JOptionPane
: It makes it easy to pop up a standard dialog box.

JPanel
: It is a generic lightweight container.

JPasswordField
: It allows editing of a single line of text where the view does not show the original characters.

JPopupMenu.Separator
: It is a pop-up menu-specific separator.

JPopupMenu
: It is a pop-up menu.

JProgressBar
: It is a component that displays an integer value within an interval.

JRadioButton
: It is a radio button that can be selected or deselected, displaying its state visually.

JRadioButtonMenuItem
: It is a radio button menu item.

JRootPane
: It is the fundamental component in the container hierarchy.

JRun
: JRun is an Application server from Macromedia that is based on Oracle's Java Enterprise Edition.

JScrollBar
: It is an implementation of a scrollbar.

JScrollPane
: It is a container that manages a viewport, optional vertical and horizontal scrollbars, as well as optional row and column heading viewports.

JSeparator
: It is a menu separator.

JSlider
: It is a component that lets the user select a value by sliding a knob within an interval.

JSplitPane
: It divides two components.

JTabbedPane
: It lets the user switch between a groups of components by clicking tabs.

JTable
: It presents data in a two-dimensional table format.

JTextArea
: It is a multiline area that displays plain text.

JTextField
: It allows the editing of a single line of text.

JTextPane
: It is a text component that can be marked up with attributes.

JToggleButton.ToggleButtonModel
: It is the toggle button model.

JToggleButton
: It is a two-state button.

JToolBar.Separator
: It is a toolbar-specific separator.

JToolBar
: It is a toolbar, useful for displaying commonly used controls.

JToolTip
: It displays a tooltip for a component.

JTree.DynamicUtilTreeNode
: It can wrap vectors/hashtables/arrays/strings and create appropriate children tree nodes.

JTree.EmptySelectionModel

It is a tree selection model that does not allow anything to be selected.

JTree

It displays a set of hierarchical data as an outline.

JViewport

It is the viewport through which you see the information.

JWindow

It is a window that can be displayed anywhere on the desktop.

List

It extends Collection to implement lists of objects.

Lock

A lock is created so as to control the access of multiple threads over the same resource. It provides exclusive access to a shared resource, which means that it allows only one thread to access an object at a time.

Map

It cannot contain duplicate keys; each key can map to at most one value. This interface maps keys to values.

Method

This is a function built into a class or an object. You can have instance methods and class methods. You use instance methods with objects, but you can use a class method just by referring to the class by name—no object is required.

Multicast

Unlike unicast, multicast is an identifier for a set of interfaces (typically belonging to different nodes).

Multithreading

This feature helps you write interactive programs, wherein multiple tasks can be performed simultaneously, thus making it a robust programming language.

Object

This is an instance of a class, much as a variable is an instance of a data type. You can think of a class as the type of an object, and you can think of the object as an instance of a class. Objects encapsulate methods and instance variables.

Permit

A permit is like a token to a lock that allows the thread to access a resource. A thread must have a permit from the semaphore before obtaining a resource. When the thread stops, it returns the permit to the semaphore, allowing another thread to access next.

Platform Independent

It is one of the most important features of Java as it is the first programming language that is not bound to any specific operating system.

PROPERTIES

It indicates that the properties of the image are now available.

Queue

It extends Collection to handle special type of lists.

Selectors

Selectors enable you to perform I/O through multiple channels.

Set

It extends Collection to implement sets, in which all elements must beunique.

It indicates that more pixels for drawing the scaled version of image are available.

SortedMap

It extends the Map to implement a sorted map.

SortedSet

It extends Set to implement a sorted set.

Synchronization

Thread synchronization, also known as serialization, is a mechanism to ensure that no two concurrently executing threads possess the same parts of a program at the same time.

Tomcat

The Tomcat server is a Java-based Web application container that was created to run Servlets and Java Server Pages in Web applications.

Unicast

As the name indicates, it is an identifier for a single interface.

WebLogic

The WebLogic Platform delivers application infrastructure technology in a single, unified, easy-to-use platform for application development, deployment, and management.

WIDTH

It indicates that the width of the base image is now available (and can be read from the width argument of the imageUpdate method).

Working directory

The current working directory of the current process is the default value; usually the system property, user.dir, imparts name to this directory.



Index

A

@Deprecated, 828, 830

@Documented, 828

@Inherited, 828-829

@Overdrive, 828

@Retention, 828-829, 842-843

@SuppressWarnings, 792, 828, 830

@Target, 828-830

AbstractCollection, 768-769, 779-782, 784, 786, 789-790, 811, 813

AbstractCollection class, 779-780

AbstractList, 768, 780-782, 786, 811, 813

AbstractList class, 780-781

AbstractMap, 767, 769, 795-796, 799

AbstractMap class, 767, 795-796

AbstractQueue, 768

AbstractSequentialList, 768-769, 781-782, 786

AbstractSequentialList class, 781-782

AbstractSet, 769, 789-790

AbstractTableModel, 477

Accelerators, 549-550, 565-566, 574

access specifier, 18-19, 114-117, 120, 148, 150-151

ActiveX, 174, 846

Adapter Classes, 147, 171, 200-202, 206, 214, 277, 320

adapter classes, 147, 171, 200-202, 206, 214, 277, 320

addElement, 471-472, 482, 495, 500, 811, 813

addition assignment, 78, 88-89

addLayoutComponent, 229, 231, 234, 236, 239, 244, 547

addSeparator, 333, 337-338, 340-343, 553, 557-559, 567-568, 570-571, 574-575, 577

AdjustmentEvent, 194, 266-270, 463-464

AdjustmentEvent class, 266-267

AdjustmentListener, 194, 265-267, 269-270, 463-464

adjustmentValueChanged, 266, 268-270, 464

ALIGN, 188, 228-229, 244, 405-406, 550

ALLBITS, 305, 544, 936

Annotations, 12, 770, 822-824, 826-831, 834, 836, 838, 840, 842-844, 890, 975

anonymous inner classes, 145, 168, 202

append, 15, 26, 29, 38, 61-62, 65, 96, 131, 164, 248-249, 260, 329, 407-409, 412, 639, 641, 648-649, 653, 658, 664, 670, 731

Appendable Interface, 164

appletviewer, 6, 173, 187, 189-190, 192, 707, 841-842, 853

Application Servers, 2, 9

Application servers, 2, 9

ArithmeticException, 71, 109, 132, 135-138

ArrayDeque, 769, 784-785

ArrayDeque class, 784-785

ArrayList, 73, 100-102, 165-167, 703, 720, 733, 768-769, 775-777, 780, 782-784, 788, 794-795, 819-820

ArrayList class, 782-783

Arrays, 13, 30-36, 38, 40, 42, 44, 46-50, 52, 54, 56-58, 60, 62, 64, 66, 68-70, 72-74, 97, 100-101, 105-106, 113, 118, 124, 126-127, 129, 144, 184, 214, 294, 313, 354, 615-616, 651, 653, 672, 767, 769, 772, 775, 779-780, 782, 801-804, 818, 821, 826-827, 830, 865, 933, 944

Arrays class, 767, 801-804

Atomic variables, 718-719

B

BartC, 733
BartQ, 733
BartS, 733
BEA Weblogic, 886
BeanBox, 846, 850-853, 855-856, 862-863, 867-868
BeanDescriptor, 857, 865, 868, 871
BeanInfo Interface, 847, 849, 856-857, 865, 871
BeanInfo interface, 847, 849, 856-857, 865, 871
Beans, 10, 386, 824, 826, 837, 844-850, 852-862, 864-874, 890
bidirectional linked list, 794
binary semaphores, 699
bitwise And assignment, 77, 89
bitwise Or assignment, 77, 89
bitwise unary Not, 77, 83
bitwise Xor assignment, 77, 89
Blackboard Assignment Retrieval Transaction, 732
Blob, 878, 880, 895, 901
Blocked, 12, 213, 675-676, 717, 923
BlockingQueue, 715-717, 719-720
Boolean, 17-18, 31, 34, 37, 39, 43-44, 46, 53-56, 61-62, 66-69, 83, 86, 133-134, 161, 167, 175-176, 178-180, 182-187, 196, 200, 203, 210, 212, 218, 224-225, 229, 253-255, 266-267, 273, 277-278, 282, 284, 286, 288, 290-291, 293-296, 300-301, 305-310, 324-326, 332-334, 339-340, 345-346, 349, 359-366, 368, 370, 373, 375-376, 380-382, 387-388, 391, 400, 404, 407-410, 419-422, 430, 433, 437, 447-448, 452, 456-461, 463-464, 466-468, 472, 475, 477, 480, 483, 485, 492, 494, 500-502, 510-511, 514-516, 519, 529, 535-537, 552-554, 556, 561-563, 571-572, 574-575, 581-582, 585, 588, 590, 592-595, 600-601, 610-611, 617, 620-624, 627, 629-630, 632, 635-636, 639-640, 642, 644-648, 650-652, 655, 657-658, 660, 662-664, 685, 691, 696-698, 705-706, 708, 710-713, 718, 722, 737-738, 740-742, 748, 750, 752-754, 764-765, 770-771, 774, 776-797, 800, 802, 808, 810-812, 814-816, 836, 839-840, 848, 858-860, 862, 864, 890, 892, 894-898, 901, 904-909, 935-937, 942
Boolean Properties, 176, 359, 848, 864
bootclasspath, 22, 25
Border interface, 381-382
Border Layouts, 215, 234, 243, 245, 269, 356, 370, 372
BorderFactory class, 379-382
Box class, 346, 521-522, 543-546, 548, 580
boxing, 67
BoxLayout class, 541-542
branching statements, 78
breakpoints, 110, 141-142
Brightening, 275-276, 311-312
bubble, 246, 265, 445
Buffer class, 617-618, 654
BufferedWriter, 164, 623, 656
BufferReader, 164, 620
business logic, 9, 970-971
ButtonGroup, 389, 432, 437-438, 563
byte, 17, 33-34, 37, 42, 45-47, 53-54, 66, 94, 164, 176, 191, 207, 284, 287, 293, 309-310, 401, 616, 618, 620, 623-624, 630-639, 643, 650-651, 654, 662-664, 737, 740-741, 743, 746, 757-758, 801-803, 840, 901, 905-906, 933-934
Bytecodes, 3-4, 6, 19-20, 45

C

Caching, 445, 478, 725, 728-729, 736, 752, 766
CallableStatement, 877, 879-880, 894, 896, 913
Canvas, 172, 276, 303-304, 354, 709, 854, 860, 862-863, 866, 932, 969, 971, 990-991
canvas, 172, 276, 303-304, 354, 709, 854, 860, 862-863, 866, 932, 969, 971, 990-991
Canvas class, 303-304, 854
Card Layouts, 215, 236, 243
Cascading, 564, 970-971
Catalog, 101, 166, 894-895, 909
catch block, 109-110, 133-136, 138-140, 716
ChangeListener interface, 458
Channels, 616, 618-620, 734, 825
Character, 9, 31, 34, 37, 39-40, 42-43, 52-55, 57-59, 62-63, 66-69, 110, 113, 126, 131, 163-164, 191, 218-219, 249, 280-282, 284-285, 287, 293, 397, 400, 406-407, 409, 414-415, 419, 427, 494, 515, 529, 615-616, 631-633, 635-636, 640-644, 647-648, 650-660, 665, 670, 672, 730, 732, 743, 754, 803, 825, 837, 877, 880, 901, 905-906
CharArrayWriter, 164, 651, 653-654
CharBuffer, 163-164, 617

CharBuffer class, 163
Charsets, 620, 799
CheckboxGroup class, 227
CheckboxMenuItem, 336-341
Choice, 2, 15, 97, 101, 172, 195, 205, 245-246, 260-265, 274, 393, 406, 434, 511, 603, 696, 978
Choice class, 261
class variables, 105, 107-108, 111, 114-115
ClassLoader, 410, 628, 677, 685, 763
ClassNotFoundException, 109, 134
clipRect, 293, 710-712
Clob, 878, 880, 895, 901
Collection, 5-6, 8, 25, 36, 66-67, 70, 96, 99-102, 105, 123-125, 130, 142, 144, 148, 161, 166, 294, 306, 354, 412, 522, 616, 626-627, 697, 715, 719, 723, 727, 731, 736, 763, 767-769, 772-784, 786, 788-791, 793, 796-797, 801, 811-812, 814-816, 820, 822, 825, 857, 871, 880, 970
Collection interface, 100, 767-768, 776-777, 779, 822
Color class, 190, 300-301
Combo Box Model, 489, 492, 500
Combo Boxes, 172, 488-492, 494, 496-500, 502, 504, 506, 508, 510, 512, 514, 516, 518, 520, 549-550, 576, 578
ComboBox, 172, 475-476, 493, 495-499, 534
ComboBoxEditor interface, 498
Comparator, 52, 767, 778-779, 783, 790-792, 799-801, 804, 821
compareTo, 53, 91, 620, 644, 793, 808-809, 820-821
ComponentEvent, 181, 194, 277, 280, 509, 587
ComponentListener, 175, 177, 181, 194, 508-509
ComponentListener interface, 508-509
ComponentResized, 509, 587
componentResized, 509, 587
Concatenating Strings, 32, 58
Concurrency Utilities, 673, 718-719, 723
Concurrent collections, 719
conditionals, 11, 75-78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102, 104
Connected Device Configuration (CDC), 8
Connected Limited Device Configuration (CLDC), 8
Connection interface, 875, 893-896, 899, 912, 914
Connection management, 877, 879
Connection.close, 923-924
Constrained Properties, 845, 859, 868-869, 872, 874
Container class, 166, 183-186, 244, 354-356, 358, 371, 373
ContainerEvent, 185, 194
ContainerListener, 184-185, 194
cookie, 3, 108, 734
CountdownLatch, 696, 701-703, 720
CountDownLatch Class, 701-702
counting semaphores, 699
createTitledBorder, 381-382, 435, 438-439, 545-546, 548
currentThread, 678, 682, 684, 686-687, 689-690, 692, 694-695, 705
Custom Class Loader, 677
Custom List Cell Renderer, 443, 472, 488
Customizers, 847, 849
CyclicBarrier, 696, 699-701, 721
CyclicBarrier Class, 699-700

D

Data Types, 31, 34, 36-37, 40, 42, 45-46, 52, 66, 74, 112, 129, 478, 745, 775, 782, 879-881, 904-906, 908-909, 919
Database Access, 879, 881
Database Metadata, 446, 879, 881
DatabaseMetaData, 877, 881, 894, 909, 921-922
DatabaseMetaData.ownDeletesAreVisible, 922
Datagram, 726, 729, 732, 734-735, 756-758, 766
DatagramPacket, 726, 756-758
DatagramPacket class, 756-757
DatagramSocket, 618, 726, 740, 756-759
DataSource, 881-882
debugging, 22, 106, 110, 141, 143-144, 355, 360, 954, 956
DEFAULT_LAYER, 369, 526
DefaultComboBoxModel, 495, 500
DefaultComboBoxModel class, 495, 500
DefaultMutableTreeNode, 475, 484-487
defaultsProperty, 385
Delegated Classes, 197
deleteCharAt, 62, 65

Deployers, 2, 10
Deque, 720-721, 768-769, 784-786
Design Patterns, 845-849, 862, 864-865, 868
Desktop Panes, 578-580, 582, 584, 586, 588, 590-592, 594, 596, 598, 600, 602, 604, 606, 608, 610, 612, 614
destroy, 171, 186, 190, 214
Dialog class, 316-317, 344-346
Dialog Modality, 209, 213
Dictionary class, 814-815
Disabled icon, 419-420, 426, 528-529
Disabled Images, 396, 426
DISPOSE_ON_CLOSE, 377, 517-518, 589
Distributed Notification, 845, 873-874
Distributed Transaction, 881-882
Divider Size, 521, 540-541
division assignment, 77, 89
division operators, 83
do-while, 17, 75, 96-97
DO_NOTHING_ON_CLOSE, 377, 429
Domain Name System (DNS), 743
Double Buffering, 276, 673, 713-715, 724
double buffering, 276, 673, 713-715, 724
Double Clicks, 256, 443, 472-473, 488
DRAG_LAYER, 369, 526
drawArc, 191, 293, 299
drawBytes, 191, 293
drawChars, 191, 293
drawImage, 191, 205, 270, 290-293, 305, 308, 311-313, 708, 711-712, 714, 930, 934-936, 938-943, 946, 948
drawLine, 191, 205, 293, 297, 299, 668
drawOval, 191, 294, 297-298
drawPolygon, 191, 294, 299
drawPolyLine, 191, 294, 299
drawRect, 191, 294, 297-298, 327-328, 668, 714
drawRoundRect, 191, 294, 297, 299
drawString, 146, 173-174, 187-188, 190-192, 201-203, 206-207, 212, 264, 269, 289, 294, 374, 376, 378, 382, 589, 714, 841, 854, 856, 861, 867, 936, 938
Driver interface, 875, 877, 879, 892, 926
DriverManager, 14, 875-879, 883, 892-893, 899, 903, 912-913, 926, 956, 966-968
DriverManager class, 14, 875, 878, 892-893, 912-913, 926
DropMode class, 474
dual state, 216
Dynamic Link Libraries (DLLs), 3
dynamic method dispatch, 145, 157

E

Editable Combo Boxes, 489, 497
EditorKit, 409-411
elevator, 246
Embossing, 275, 313-314
Empty String, 32, 56, 60, 248, 346, 407, 732
encoding, 21, 53, 110, 642-643, 651, 735, 752, 798-799, 986
Enum, 14, 17, 99, 177, 359, 473-474, 773, 805-809, 819, 830, 877-878
Enum Constant, 473-474, 808-809
Enumerations, 166, 767, 804-805, 808-809, 826
equalsIgnoreCase, 54, 91, 320, 328, 330, 336, 738, 760
ERROR, 21, 27, 38, 45-47, 51, 71, 73, 87, 109-110, 115, 130, 132-133, 135-136, 138, 151, 159-160, 163-164, 207, 305-307, 367, 406, 423, 429, 513, 516, 544, 599, 605, 627-628, 640, 657-659, 726, 749, 761, 805, 809-810, 829-830, 868, 881, 893, 908, 926, 938-939, 945-946, 966
Events, 142, 147-148, 171, 174-176, 181, 184-185, 194, 197-200, 204, 206, 209, 216, 218-219, 222, 225, 246, 253-255, 261-262, 265-267, 269, 273, 278-279, 315, 318, 320-325, 329, 334-335, 340, 350, 356, 362, 375, 377, 397, 404, 419, 421, 426-427, 441, 444, 472, 486, 489-490, 494, 496-497, 504-505, 508, 553-554, 593, 611, 621, 626, 722, 824, 835-836, 845-849, 852, 864-865, 867, 871-874, 882, 926, 976-977
EventSetDescriptor, 849, 857, 865, 871
exception handling, 109-110, 144, 877, 881
exception subclasses, 891
Exchanger, 696, 702-703, 721
Exchanger Utility, 702
executeUpdate, 897, 903-904, 913, 917, 924-925
EXIT_ON_CLOSE, 212, 374, 377-379, 384, 406, 423, 435, 438, 475, 478, 480, 531, 584-586, 588, 941
Extension classes, 22, 28, 677

F

faultyGenericMethod, 73-74
FeatureDescriptor, 858-859, 865-866, 871
FeatureDescriptor class, 858
File Chooser Filters, 489, 518
File Dialog Boxes, 315-316, 348-349
file servers, 729
File Transfer Protocol (FTP), 726
FileChannel, 618-619, 639, 649
FileDialog, 316-317, 348-349
FileDialog class, 316-317, 348-349
FileFilter class, 518-519
FileInputStream, 136, 138-139, 618-619, 637-638, 665
FileOutputStream, 207, 618-619, 634, 638-640, 665
FileReader, 164, 647-648, 655, 661
Files Class, 620, 622-624
FileStore Class, 628-629
FileSystem Class, 626-627
FileSystems Class, 627-628
FileSystems.getDefault, 620, 626
FileWriter, 164, 648-649, 654
Filtering, 444, 478, 480-482, 773, 933
FilterWriter, 164
finalize, 105, 124-125, 144, 161, 294, 637, 639, 808, 839
Finally, 13, 15, 17, 20, 27, 29, 86, 100, 106, 123, 133-134, 137, 186, 204, 244, 258, 330, 356, 406, 473, 484, 486, 522, 664, 698, 718, 768, 807, 834, 846, 854, 862, 899-900, 945, 947, 953, 968, 992
fireTableChanged, 482-483
Flicker, 673, 710-711, 714-715, 947
Floating-Point Literals, 31, 38
Floating-point numbers, 17-18, 33-34, 37-38
flow layout manager, 202, 216, 228-229, 231, 318
FlowLayout class, 228-229
FocusEvent, 179-181, 194, 196-198
focusGained, 196-198
FocusListener, 175, 181, 194, 196, 198
focusLost, 196-198
font, 177, 179, 182, 186, 276, 283-288, 294-295, 360, 363, 381, 384-385, 387, 391-392, 398, 404, 408-409, 414-416, 422-423, 457, 466, 569, 576, 824, 868, 986
Font class, 283-286
FontMetrics class, 286-288
for-each, 70, 72, 76, 99-101, 148, 164, 166-167, 774, 798, 807, 891
Formatter, 164, 615, 666, 668-672
Formatting Numbers, 32, 60
Foundation Classes, 211, 352, 394
Frame class, 146, 174, 202-203, 316, 318, 330, 332, 355, 374
FRAMEBITS, 305, 544

G

garbage collection, 5, 25, 105, 123-125, 142, 144, 161, 294, 306, 825
Generic Class, 784, 788
getAdjustable, 267-271
getAllByName, 737-739
getBorderInsets, 381-382
getByName, 737, 739, 760
getChannel, 618-619, 639, 649, 741, 748
getChildCount, 475, 483-486
getClickCount, 473
getColumnLabel, 482-483, 909, 968
getData, 120-121, 126, 128, 158-159, 757-758, 864
getDividerSize, 536, 540-541
getFileName, 620-621
getFileStores, 626-628
getHeight, 177, 205, 287-288, 290, 309, 360, 379, 585, 587, 862, 931, 934, 941-943
getHorizontalAlignment, 399, 404-405, 419
getKeyCode, 280-281
getLayoutAlignmentX, 234, 236, 239, 244, 542, 547
getLayoutAlignmentY, 234, 236, 239, 244, 542, 547
getListCellRendererComponent, 472, 500
getModifiers, 222, 278, 280, 343, 573
getName, 13, 178, 285, 390, 515-516, 621, 645-647, 678-679, 682, 685-687, 689-690, 692, 694-695, 705, 764, 820-821, 839-840, 848, 858, 942
getOpacity, 324, 586
getParent, 178, 343, 483, 486, 620-621, 645-647
getPath, 475, 516-519, 620, 626-627, 645-646, 750, 942
getPathMatcher, 626-627

getPreferredSize, 178, 180, 185, 205, 218-219, 249, 254, 360, 404, 408, 410, 508-509
getPreferredSize(), 178, 180, 185, 205, 218-219, 249, 254, 360, 404, 408, 410, 508-509
getQuote, 733, 760
getRoot, 620-621
getRootDirectory, 626
getSelectedCheckbox, 227
getSelectedIndices, 466, 468, 470
getSelectionEnd, 251
getSelectionStart, 251
getSize, 178, 183, 212, 269, 271, 285, 288, 327-328, 360, 415, 495, 509, 714, 840, 854, 856, 861, 867, 943, 947-948
getState, 203, 224, 226-227, 337, 340-341, 562
getUserPrincipalLookupService, 626-627
getViewPosition, 448, 450
Gif Writer, 209, 213
Global addresses, 736
Glue, 522, 543-546
goto, 17, 103
Graphics class, 187, 190, 276, 290-295, 298-299, 710, 940
Grayscale, 275-276, 312-314
Grid Bag Layouts, 215, 238, 243
Grid Layouts, 215, 231, 243
GridBagConstraints class, 240-241
GridBagConstraints.BOTH, 242

H

has-a relationship, 160
HashMap, 767-769, 774, 796-798
HashMap class, 767, 796-797
HashSet, 100, 198, 768-769, 774, 778, 789-792, 821
HashSet class, 769, 789-791
Hashtable class, 774, 814-817
HIDE_ON_CLOSE, 377
HORIZONTAL_SCROLLBAR_ALWAYS, 408, 453, 455
HORIZONTAL_SCROLLBAR_AS_NEEDED, 453
HORIZONTAL_SCROLLBAR_NEVER, 453
Host Name Resolution, 736
HSPACE, 188
HyperlinkListener, 398, 410
Hypertext Transfer Protocol (HTTP), 726

I

IDS Driver, 887
if-else Ladders, 75, 93
IllegalArgumentException, 109
Image class, 276, 289-290, 927-928, 930-931, 934-935, 939-941, 944
ImageIcon, 385-386, 391, 397, 401-403, 413, 424-425, 427, 431-432, 440, 449-450, 455, 472, 500, 530, 532-533, 559, 573, 575, 577, 927-928, 938
imageIcon, 385-386, 391, 397, 401-403, 413, 424-425, 427, 431-432, 440, 449-450, 455, 472, 500, 530, 532-533, 559, 573, 575, 577, 927-928, 938
ImageIcon class, 397, 401-402, 424, 927-928, 938
ImageObserver, 176, 180, 275, 290-291, 293, 305-306, 314, 402, 483, 927, 930-932, 934-935, 937, 941-943
imagePainter, 385
imageUpdate, 179, 305, 400, 420, 935-936
Improved maintainability, 719
Increased performance, 719
Increased productivity, 719
Increased reliability, 719
Indexed Properties, 848, 864-865
IndexedPropertyDescriptor, 865, 871
indexOf, 54, 59, 62, 252, 778, 781-782, 787, 812
InetAddress class, 736-737, 739-740
inner class, 147, 168, 202, 323, 377, 769
Inner Frames, 579-580, 582, 584, 586, 588, 590, 592, 594, 596, 598, 600, 602, 604, 606, 608, 610, 612, 614
InputEvent, 195, 277-278, 280, 566, 573
InputEvent.ALT_DOWN_MASK, 566
InputEvent.ALT_GRAPH_DOWN_MASK, 566
InputEvent.CTRL_DOWN_MASK, 566
InputEvent.META_DOWN_MASK, 566
InputEvent.SHIFT_DOWN_MASK, 566
inputMap, 360, 363, 385, 391
InputMismatchException, 109, 135, 137
InputStreamReader, 109-110, 164, 615, 641-642, 647, 660, 672, 742-743, 749

INSERT_ ROWS, 474
INSERT_COLS, 474
insertString, 412-413
Insets, 184-185, 215, 241, 243, 351, 360, 379, 381-382, 387, 394, 419, 421, 448, 551-552, 571, 574-575
installBorder, 391-392
installColorsAndFont, 391-392
instance variables, 105, 107-108, 111, 114, 116, 144, 807, 849, 870, 944, 946
Integers, 34, 37, 42, 83, 129, 310, 525, 825, 880, 933
Internet Addressing, 730
Interpreter, 5-6, 8-9
introspection, 846-847, 849-850, 863, 865
intValue, 66-68, 525-526, 703
invalidateLayout, 234, 236, 239, 244, 542, 547
IOException, 109-110, 130, 134, 138-140, 214, 411, 413, 475, 619, 630-634, 636, 639-642, 648, 652, 654, 744-747, 751, 761
IP Port, 728
ipadx, 241, 243
ipady, 241, 243
is-a relationship, 160
isAlive, 685, 689-690
isfocuable, 197
isLeaf, 483, 485
isPasswordCorrect, 406-407
isReachable, 737
Item Event, 263
ItemEvent class, 225, 262-263, 497
ItemListener, 194, 205, 224-227, 249, 253-255, 258-259, 261-264, 266, 336, 340, 418-420, 433-434, 437, 440-441, 493-494, 496-497, 563, 576
itemStateChanged, 205, 225-227, 249, 259, 262-264, 337, 340, 434, 437, 440-441, 497, 563, 577
Iterator, 72, 101-102, 164-167, 294, 621, 767, 773, 776, 778-787, 790-795, 797, 811
Iterator object, 72, 166

J

JApplet class, 355, 364-366, 374, 710
jar, 6, 14, 22-25, 28-29, 211, 620, 627, 671, 735, 799, 822-826, 828, 830, 834, 836-842, 844-845, 853-855, 862, 957, 959, 980
JarEntry class, 839-841
JarFile class, 838-839
Java API, 6, 8, 16, 66, 671, 824-826, 844, 883
Java applets, 4-7, 173, 928, 939
Java Consoles, 171, 191
Java Database Connectivity (JDBC), 7, 876, 952
Java Development Kit, 6, 15, 889, 970
Java exception classes, 134
Java Micro Edition (Java ME), 7
Java Native Interface (JNI), 26, 810, 885
Java Platforms, 6-7
Java syntax, 2, 11, 33, 79, 107, 172
Java Virtual Machine (JVM), 3, 665, 952, 971
java.lang.annotation, 824, 826, 828, 830, 842-843
java.lang.instrument, 824, 826
java.lang.invoke, 824, 826-827
java.lang.management, 825-826
java.lang.Math class, 89-90, 109
java.lang.ref, 825-827
java.lang.reflect, 12, 825-827, 842-843
java.net, 8, 597, 725-727, 734-735, 737-739, 742-744, 746, 749, 751, 754-756, 758-760, 825, 880, 902, 987
java.net package, 725-726, 734-735, 737, 744, 756, 880
java.security package, 725, 761-764, 766
java.security.Permission, 764
java.security.Policy class, 765
java.sql package, 875, 879-881, 904, 911
java.sql.ResultSet Interface, 904-907
java.util.concurrent.atomic, 13, 673, 719, 722-723
java.util.concurrent.locks, 13, 673, 719, 721-722
javac, 6, 9, 12, 20-23, 25, 38, 46, 73, 99, 115, 136, 141, 159-160, 163, 188, 841, 981, 989-991
javacdoc, 6
javah, 6
javap, 6
JavaSoft Framework, 888-889
JButton, 353, 366, 368, 383, 389, 393, 413, 417, 421-425, 427-429, 434-435, 438-439, 449, 461, 481, 504-506, 511, 516, 518, 531-534, 538-540, 569-570, 574-577, 582, 584-586, 588, 596-597, 604-605, 607-609, 612-613, 941
JCheckBox, 353, 393, 433-436, 438-441, 534

JCheckBoxMenuItem, 353, 560-562

JCheckBoxMenuItem class, 560-562

JColorChooser, 353, 510-511

JColorChooser class, 510-511

JComboBox, 353, 475-476, 492-496, 498-499, 534, 577

JComboBox class, 492-495

JComponent, 353-355, 358-364, 366, 368-370, 373, 388, 390-391, 397, 399, 403, 417, 421-422, 430, 433, 436, 447, 451, 456, 462, 464-465, 473, 492, 501, 506-507, 510-512, 514-515, 527-528, 535, 543, 551-552, 555, 560, 562, 570, 574, 590-591, 599, 709

JComponent class, 354-355, 358-364, 397

jdb, 6, 110, 141, 143

JDBC API, 876-883, 886, 889, 892, 908-909, 923, 926

JDBC Architecture, 877-878, 883, 889

JDBC Driver Manager, 889

JDBC Driver Test Suite, 889

JDBC DriverManager, 876

JDBC test suite, 876

JDBC-ODBC Bridge, 14, 876, 884-885, 889, 913, 956

JDBC-ODBC bridge, 14, 876, 884-885, 889, 913, 956

JDesktopPane, 353, 580, 590-591, 593, 596-598, 600, 609

JDesktopPane class, 580, 590

JDialog, 353, 355, 370, 510, 579-580, 583, 600, 609-614

JDialog class, 580, 609-611, 613

JEditorPane, 353, 397-398, 409-412

JFileChooser, 353, 512-518, 941-942

JFileChooser class, 512-516

JFrame, 212, 351, 353-356, 366, 370-372, 374-379, 383-384, 394, 406, 412, 415, 423, 428-429, 434-435, 438, 475, 478, 480, 485-486, 491, 516-518, 522, 531, 580-581, 583-589, 926, 941

JGoodies, 356

JInternalFrame, 353, 370, 580, 590-598, 600

JInternalFrame class, 580, 591-595

JInternalFrame.JDesktopIcon, 353, 592-594

JLabel, 353, 383-384, 392-393, 397, 399-400, 402-403, 422-423, 449-450, 455, 464, 472, 475, 481, 500, 523-526, 530, 612-613

JLayeredPane, 353-354, 356, 365-370, 375-376, 525-526, 580-582, 590, 593-595, 611

JLayeredPane.PALETTE_LAYER, 525-526

JList, 353, 393, 445, 450, 465-474, 484, 500

JList class, 445, 465-468

JMenu, 353, 550-554, 557-559, 561, 563-564, 566-570

JMenu class, 552-554, 557, 569

JMenuBar, 353-354, 356, 365, 367-368, 375-376, 550-552, 557-561, 563-570, 593, 595, 611

JMenuBar class, 551-552, 557-558

JMenuItem, 353, 552-560, 562-563, 566-571, 573

JMenuItem class, 555-557

Join, 12, 58, 681, 685, 688-690, 692-694, 696, 701, 705, 707

Joining Threads, 673, 688

JOptionPane, 210, 353, 406, 423-424, 429, 435, 439, 579-580, 598-609, 836

JOptionPane class, 580, 598-603, 605-606, 608

JPanel, 353-354, 371-374, 376, 378, 382-383, 389, 393, 423, 429, 435, 438-439, 449-450, 453-454, 464, 475, 481, 511, 522, 530-532, 534, 538-540, 542, 545-546, 548, 585, 589, 595-597, 838, 936, 938, 941-943

JPanel class, 354, 373, 530

JPasswordField, 353, 397, 406-407

JPopupMenu, 353, 360, 363, 553, 570-573

JPopupMenu.Separator, 353

JProgressBar, 353, 500-505

JProgressBar class, 500-502

JRadioButton, 353, 389, 393, 436-439, 534

JRadioButtonMenuItem, 353, 562-563

JRootPane, 353-354, 356, 360, 364-368, 375-376, 391, 581-582, 592-593, 595, 610-611

JRootPane class, 356, 364, 366-368

JRun, 9-10

JScrollBar, 353, 393, 445, 451-452, 462-464

JScrollBar class, 445, 462-464

JScrollPane, 353, 397, 407-409, 411, 413, 450-455, 468, 470-471, 473, 475, 479, 481, 483-485, 487, 499

JSeparator, 353, 507-509

JSeparator class, 507

JSlider, 353, 445, 456-461

JSlider class, 445, 456-460

JSplitPane, 354, 535-541

JSplitPane class, 535-537
JSplitPane.HORIZONTAL_SPLIT, 539-541
JSplitPane.VERTICAL_SPLIT, 537-540
JTabbedPane, 354, 527-534
JTabbedPane class, 527-530
JTable, 354, 445, 473-474, 476-479, 481, 483
JTextArea, 354, 397, 407-409, 415-416, 595-596, 598
JTextField, 354, 391-393, 397, 403-407, 423-426, 428, 453-454, 474-475, 481, 486-487, 506, 508-509, 516, 518, 537-540, 542, 545-546, 548, 573, 589, 613
JTextPane, 354, 397-398, 409, 412-416
JToggleButton, 354, 398, 430-433, 436
JToggleButton.ToggleButtonModel, 354
JToolBar, 354, 550, 574-577
JToolBar.Separator, 354
JToolTip, 354, 359, 506
JToolTip class, 506
JTree, 354, 446, 473-476, 483-487
JTree class, 446, 483
JTree.DynamicUtilTreeNode, 354
JTree.EmptySelectionModel, 354
Just-In-Time (JIT) compiler, 4
JViewport, 354, 404, 445, 447-452
JWindow, 354-355, 370, 580-583, 604

K

key set, 620
KeyEvent, 179-181, 195, 198, 279-282, 289, 362, 494, 552, 554, 556, 566, 572
KeyListener, 175, 178, 181, 194, 279, 282, 288
KeyListener interface, 279
keyPressed, 279-280, 282, 289
keyReleased, 279-280, 282, 289
KeyStroke class, 565
keyTyped, 279-280, 282, 289

L

Labels, 33, 172, 213, 215, 220-221, 238, 274, 395, 397, 399, 401-403, 424, 442-443, 454, 457-458, 460-461, 488, 490, 523-526, 530-531
Layered Panes, 368, 521-522, 524-526, 528, 530, 532, 534, 536, 538, 540, 542, 544, 546, 548
LayoutPath class, 286
Layouts, 214-218, 220, 222, 224, 226, 228, 230-232, 234, 236, 238, 240, 242-245, 269, 356, 370, 372, 520-522, 524, 526, 528, 530, 532, 534, 536, 538, 540-546, 548
Life Cycle, 675-676
Link-local addresses, 736
LinkedList, 715, 768-769, 777, 781, 784, 786-788, 793-794
LinkedList class, 786-788
List Interface, 767-768, 776-778
ListIterator, 165, 767, 778, 781-782, 787, 793-795
ListModel, 465-467, 478
ListSelectionListener, 465, 468, 470, 472
Literals, 31, 37-43, 57, 72, 74, 112, 118
locationToIndex, 467, 473
Locks, 13, 673, 699, 718-719, 721-723, 893, 895
logical And, 75, 77, 85, 87
logical Or, 77, 87
logical unary Not, 77, 83
logical Xor, 77
loops, 11, 17, 70, 75-80, 82, 84, 86, 88, 90, 92, 94, 96-100, 102-104, 148, 703

M

Map Interfaces, 769
Map.Entry, 769, 796-798, 800-801
Marker, 27, 761-762, 765, 823, 828-829, 842-843, 849, 870
Matcher.find, 481
Math Class, 51-52, 75, 78, 89-90, 92, 104, 109
MAX_PRIORITY, 684, 690-691
MediaTracker Class, 275, 305-307, 314, 927, 937-938
Memory Management, 4-5, 7, 25, 105, 123-124, 144
MemoryImageSource, 275, 308-313, 933
Menu class, 316, 330, 332-333
Menu Separators, 315, 336, 338, 350, 557
MenuBar, 203, 315-316, 330-333, 335-341, 347, 349, 365, 367, 376
MenuItem, 210, 315-316, 330, 332-343, 347, 349
MessageFormat class, 70
Metal look and feel, 388, 390
MethodDescriptor, 849, 857-858, 865-866, 871

Microsoft Foundation Class (MFC), 3
MIN_PRIORITY, 684, 690-691
mnemonic, 71, 399-400, 418-419, 421, 427-428, 513-515, 529, 555, 566
MODAL_LAYER, 369, 526
Model View Controller, 356
modulus, 75, 77, 83, 89, 96
Motif look and feel, 388, 390
MOUSE_OVER, 385
MouseAdapter class, 201-202
mouseClicked, 201-202, 209, 238, 277-278, 292, 297, 303-304, 344, 473, 573
MouseEvent, 180-181, 195, 201-202, 238, 277-279, 292, 296-297, 299, 303-304, 343-344, 360, 362, 473, 529, 552, 556, 572-573, 715, 854, 860, 867
MouseEvent class, 277-278
mouseless operation, 355
MouseListener, 147, 163, 175, 178, 181, 194, 201, 237-238, 277-279, 291, 296, 302, 304, 343, 362, 472, 556, 573
MouseMotionListener, 147, 163, 175, 178, 181, 194, 277-279, 296, 714
MouseWheelEvent, 195, 197, 273
MouseWheelListener, 175, 178, 181, 194, 196
moveToBack, 369-370, 525, 594
moveToFront, 369-370, 525, 594
Multi-Dimensional Arrays, 31, 47-50
Multicast, 734-737
MulticastSocket, 726, 735
Multilevel Inheritance, 145, 153, 170
Multiple List Selections, 443, 469
Multiple Threads, 130-131, 672-674, 676-678, 680, 682, 684, 686, 688, 690, 692-694, 696, 698, 700-702, 704, 706, 708, 710, 712, 714, 716, 718, 720, 722, 724
MULTIPLE_INTERVAL_SELECTION, 469-470
multiplication assignment, 77, 88-89
multiplication operator, 44, 83-84
multitasking, 674
Multithreading, 4-5, 276, 674, 676, 718, 890, 975
multithreading, 4-5, 276, 674, 676, 718, 890, 975
MutableTreeNode, 483-484
mutexes, 699, 719

N

Nanosecond-granularity timing, 719
Napkin, 356
narrowing conversion, 46
NavigableMap, 720, 769, 800
Network Information Service (NIS), 736, 952
network servers, 729
Networking, 8, 724-730, 732, 734, 736, 738, 740, 742, 744, 746, 748, 750, 752, 754, 756, 758, 760, 762, 764, 766, 773, 825, 956
New, 2-3, 5, 11-15, 17-21, 23, 26-29, 34-36, 40-41, 43-50, 53-61, 63-68, 70, 72-73, 84, 90, 96, 99-102, 108-117, 119-141, 144, 146-170, 173-174, 182-183, 187-188, 191-193, 195-214, 216-221, 223-230, 232-237, 242-243, 246, 248-253, 255-258, 260-265, 267-274, 279, 282-284, 286, 288, 290-293, 295-297, 299-304, 307-308, 310-313, 316, 318-324, 326-343, 346-347, 349, 354-355, 362, 369-371, 373-379, 381-386, 389, 392-393, 397, 399-400, 402-403, 405-409, 411-418, 421-432, 434-435, 437-441, 445, 449-456, 458-461, 464, 468-475, 477-482, 484-487, 490, 495-500, 502-512, 516-518, 522-526, 530-542, 545-546, 548, 550-554, 557-561, 563-564, 566-577, 580, 582-591, 595-598, 600, 604-605, 607-609, 612-613, 616, 619-620, 622-623, 625, 627-629, 632-634, 636-642, 644, 646-649, 651-657, 659-661, 664-668, 670-671, 674-677, 680-684, 686-696, 698-720, 723, 727-728, 733, 738, 742-749, 751, 754-755, 758-760, 770, 773-776, 778-779, 781, 783-785, 788-790, 792-797, 799, 804-805, 807, 813-816, 818-821, 826, 828, 831, 833-834, 836-843, 845-847, 852-856, 859-864, 866-870, 872-873, 876, 882, 891, 895-896, 906, 910, 914-915, 917, 919-923, 925, 930, 933, 936-938, 941, 943-945, 947
newByteChannel, 623
newWatchService, 626-627
Nimbus, 351, 392-394
NORM_PRIORITY, 684, 691
Normal icon, 426
not equal to, 77, 85, 400, 420
notify, 161, 310, 704-706, 711-712, 717-719, 722, 871
notifyAll, 161, 704, 722

O

Object-Oriented Programming, 2-3, 105-110, 112, 114, 116, 118, 120, 122, 124, 126, 128, 130, 132, 134, 136, 138, 140, 142, 144, 146, 784, 788

objectCollArray, 73
OCI (Oracle Call Interface), 886
ON_OR_INSERT_COLS, 474
ON_OR_INSERT_ROWS, 474
One-Dimensional Arrays, 31, 47-50, 313
One-Touch Expandable, 521, 535, 538-540, 548
opaque, 179, 300, 325, 361, 381, 384-385, 524, 584, 586-588, 623, 732, 762
Open Database Connectivity (ODBC), 7, 876, 953
operand, 47, 78, 82, 84-87, 89, 771
operators, 36, 74-78, 80-92, 94, 96, 98, 100, 102, 107
OutputStreamWriter, 164, 615, 643, 648, 672
Overlay Layout Manager, 521-523, 547-548
OverlayLayout class, 547
ownDeletesAreVisible, 921-922

P

Padding, 215, 241, 243-244, 669, 802
paint method, 190, 361
paintComponent, 361, 372-373, 376, 378, 382, 464, 545, 585, 589, 594, 936, 938, 943
painter, 159, 385-386
PALETTE_LAYER, 369, 525-526
Panels, 215-216, 232-234, 238, 271, 507, 510-511, 530, 542, 545, 980
PARAM NAME, 188, 191, 841, 853
PARAM VALUE, 188
ParameterDescriptor, 865-866, 871
passing a reference, 118
passing by value, 118, 126
Path Interface, 620-621
Paths Class, 624-625
pccrrn, 730
Persistence, 386, 845, 847, 849, 870, 874
PixelGrabber, 275, 308-313, 934
platform-independent, 3-5, 7, 356, 643, 876
Pluggable Look and Feel, 350-352, 354-356, 358, 360, 362, 364, 366, 368, 370, 372, 374, 376, 378, 380, 382, 384, 386, 388, 390, 392, 394
Pluggable look and feel, 350-352, 354-356, 358, 360, 362, 364, 366, 368, 370, 372, 374, 376, 378, 380, 382, 384, 386, 388, 390, 392, 394
Policy Class, 763-765
Policy Tool, 207-208
polymorphism, 145, 157-158, 170
Pop-Up Menus, 209, 315, 342, 350, 522, 549, 570, 572-573
POPUP_LAYER, 369, 526
PopupMenu, 172, 175, 210, 342-343, 553-554, 571-572
postfix operators, 82
prefix operators, 82
PreparedStatement, 875, 877, 879-881, 894-896, 899-904, 913-916, 924-926
PreparedStatement Interface, 875, 896, 899-903
Pressed icon, 419, 421, 426
print servers, 729
ProcessBuilder Class, 130
Progress Bars, 488-490, 492, 494, 496, 498, 500, 502-504, 506, 508, 510, 512, 514, 516, 518, 520
PROPERTIES, 15, 29, 100, 176, 266, 285, 301, 305, 359, 383, 385, 391, 399-400, 404, 418-421, 430, 433, 436, 457-458, 463-464, 467, 493, 544, 552, 555, 560, 562, 643, 647, 728, 736, 753, 755, 763, 768, 798, 810, 817-818, 845-850, 852, 856, 858-859, 861-865, 867-874, 879, 881, 892, 900, 905, 909-911, 913-915, 917, 920-922, 925, 956, 966, 972, 979-980
Properties class, 817
PropertyChangeEvent, 359, 865, 868-870, 872
PropertyChangeListener, 175, 178, 182, 184, 210, 358, 386, 388, 404, 419, 493, 553, 571, 574, 868-869, 871-872
PropertyDescriptor class, 858-859
protocols, 6-7, 726-727, 731, 734, 752, 756, 888
Proxy Servers, 729
putClientProperty, 362, 459-461

Q

Queue, 673, 696, 715-717, 719-721, 747, 767-769, 777, 784-785, 976
Queue Interface, 673, 715, 767-768, 777

R

RadioButton, 172
Readable Interface, 163-164
ReadAllBytes, 624
readAllLines, 624
Readers, 21, 352, 616, 651
readLock, 721-722
ReadWriteLock, 721-722
reboxing, 68
Reduced programming effort, 719
registerAsParallelCapable, 677
Remote Notification, 845, 873-874
removeLayoutComponent, 229, 231, 235, 237, 240, 244, 547
replaceRange, 249, 251, 408
Replacing Strings, 32, 59
ResultSet, 477-478, 482-483, 875-877, 879, 881, 884, 894-900, 904-911, 913-926, 967
ResultSetMetaData, 477, 482-483, 875, 877, 881, 905, 908-911, 967-968
ResultSetMetaData interface, 875, 908-910
resume, 141, 680-681, 685, 705-706, 711-713, 847
RGBImageFilter, 312, 934
Rich Text Format (RTF), 397
Rigid areas, 522, 543, 545
Rollover, 396, 418-419, 421, 426-427, 440, 574-576
Rollover icon, 419, 421, 426
Rollover selected icon, 419, 421, 426
RowSetFactory interface, 877
RowSetProvider class, 877
Rowsets Object, 882
RowSorter, 478-480
rs.deleteRow, 921
RTFEditorKit, 411-412
Runnable, 211-212, 383, 425, 474, 478, 480, 533, 584-586, 588, 673-676, 680-681, 683-685, 687, 691, 695, 699-703, 705-706, 708, 710, 712-713, 720, 724, 759, 936-937, 947

S

Scanner, 109, 132, 135, 137, 164, 615, 671-672, 738, 760
scope, 19, 98, 105, 115, 122-123, 129, 144, 151, 213, 736-737, 970, 972, 992
Scroll bars, 245-250, 252, 254, 256, 258, 260, 262, 264-274, 407
scroll box, 246, 445
Scroll Pane Headers, 443, 454
Scroll Panes, 244-248, 250, 252, 254, 256, 258, 260, 262, 264, 266, 268, 270-272, 274, 443, 445, 450, 454-455, 538, 591
Scrollable, 172, 265, 408, 410, 445, 450, 483, 890, 908, 914-917, 919
ScrollBar, 172, 194, 265-270, 353, 445, 451-452, 458, 462-465, 468
ScrollPane, 172, 247, 272-274, 329, 447, 453-455, 468-470, 475, 479, 481
ScrollPane class, 247, 272-273
Security Policies, 171, 207, 214
SELECT Annotation, 890
Selected icon, 419-421, 426
selected-key set, 620
Selection Modes, 469
Selectors, 620
Semaphore, 696-699, 721, 723
ServerSocket, 618, 726, 735, 743-748, 759
Set Interface, 767, 769, 778-779
setActionCommand, 222-223, 334, 336, 404, 406, 420, 425, 494, 557-560, 569
setBackground, 182, 190-191, 193, 203, 243, 256, 268, 279, 302, 326, 344, 363, 371-373, 376, 378, 472, 500, 512, 548, 584-585, 589, 714, 863, 947-948
setBlockIncrement, 266, 268, 463
setBounds, 182, 228, 326, 363, 376-377, 379, 517-518, 523, 525-526, 582-583, 589, 612-613
setCaretPosition, 409
setCellRenderer, 467, 471
setData, 120, 757, 864
setDefaultButton, 368, 427-429

setDefaultCloseOperation, 212, 375, 377-379, 384, 406, 423, 429, 435, 438, 475, 478, 480, 517-518, 531, 584-586, 588-589, 594, 611, 941
setDisabledIcon, 400, 420, 426
setDisabledSelectedIcon, 420, 426
setDividerSize, 537, 540-541
setEchoChar, 219, 397, 406
setEndSelection, 412-413
setFocusable, 182, 196-197
setFont, 182, 186, 288, 295, 363, 404, 408-409, 414-415, 422, 457
setForeground, 183, 191, 302-303, 363, 405, 413-416, 472, 500, 503, 505
setHgap, 229, 231, 235, 237, 243
setHorizontalAlignment, 384, 400, 402, 404-406, 420, 424
setLabel, 222-223, 225, 334, 420, 572
setLayer, 369-370, 525-526, 595
setLineWrap, 408-409
setLocation, 180, 183, 205, 217, 235, 304, 326-327, 464, 527, 572, 596-597
setMajorTickSpacing, 457, 460-461
setMenuBar(), 316, 332
setMinorTickSpacing, 457, 460-461
setModal, 345-346
setName, 183, 678, 682, 685, 848, 858
setOneTouchExpandable, 537-541
setOpacity, 203, 326, 346, 583-584, 586-588
setOpaque, 364, 472, 500, 523-525
setOrientation, 266, 458, 463, 502-503, 507, 537, 539-541, 575
setPage, 398, 410-413
setPaintLabels, 458, 460-461
setPaintMode, 212, 295, 300
setPaintTicks, 458, 460-461
setPolicy, 764-765
setPosition, 369-370, 525
setPreferredSize, 183, 206-207, 279, 364, 409, 411, 508-509, 548
setPressedIcon, 421, 426-427
setResizeable, 346
setResultSet, 478, 482
setRolloverEnabled, 421, 426-427
setRolloverIcon, 421, 426-427
setRolloverSelectedIcon, 421, 426
setRowHeaderView, 452, 454-455
setRowSorter, 478-479, 481
setSelectedCheckbox, 227
setSelectedIcon, 421, 426, 440
setSelectionMode, 468-470
setState, 203, 224-227, 340, 441, 561
setTabPlacement, 530-533
setText, 147, 163, 193, 196-200, 206, 219-220, 223-226, 228, 230, 232, 235, 250, 259-260, 263, 282, 320-321, 330-331, 336-337, 339, 341-342, 347, 350, 400, 405, 411, 420-422, 424, 426, 428-429, 472, 487, 500, 506, 517-518, 567, 597, 981, 987
setUnitIncrement, 266, 268, 463
setVerticalAlignment, 400, 402, 421, 424
setVerticalTextPosition, 400, 402-403, 421, 424
setVgap, 229, 231, 235, 237, 243
setViewportBorder, 452, 454-455
setViewPosition, 449-450
setWrapStyleWord, 408-409
setXORMode, 295, 300
shift left, 78, 89
shift left assignment, 78, 89
shift right, 78, 89
shift right assignment, 78, 89
shift right with zero fill, 78
shift right zero fill assignment, 78, 89
short-circuit And, 77
short-circuit Or, 77
showConfirmDialog, 599-604
showInputDialog, 602-603, 606-608
showInternalConfirmDialog, 602, 609
showInternalInputDialog, 602-604, 609
showInternalMessageDialog, 603-604, 609
showInternalOptionDialog, 603, 609
showMessageDialog, 210, 406, 423, 429, 435, 439, 603, 605-606, 836

showOptionDialog, 603
Simple Properties, 845, 848, 862, 874
SimpleBeanInfo class, 857-858, 861, 865, 867
Single Member Annotations, 823, 843
SINGLE_INTERVAL_SELECTION, 469
SINGLE_SELECTION, 469
Site-local addresses, 736
SocketChannel, 618-619, 741
SOMEBITS, 305, 544
SortedMap, 768-769, 799-801
SortedSet, 767-768, 779, 790-791
SortedSet Interface, 767, 779, 791
Splash Screen, 209, 211-213
Split Pane Orientation, 521, 539-540, 548
Split Panes, 352, 521-522, 524, 526, 528, 530, 532, 534-536, 538-540, 542, 544, 546, 548, 591
SqlData, 877, 880
Stack, 96, 110, 122-123, 133-136, 141-142, 684-685, 727, 768-769, 785-789, 810, 813-814, 972, 976, 978
Stack class, 813-814
Statement interface, 875, 880, 896-898, 900, 903
Statement Pooling, 881-882
Static Members, 32, 51-52
Stopping Threads, 673, 690, 705
Streams, 8, 13, 130, 164, 614-616, 618, 620, 622, 624, 626, 628, 630-636, 638, 640, 642-644, 646, 648, 650-652, 654-656, 658, 660, 662, 664, 666, 668, 670, 672, 702, 745, 747, 769-770, 772-774, 826
StrictMath class, 37, 90, 104
String Class, 32, 36, 52-61, 91, 107-108, 112-113, 120, 126, 252, 663, 669
String class constructor, 53, 56
String class field, 52
String class methods, 53-56
String Length, 32, 52, 57, 64
StringBuffer class, 32, 52, 61-65, 131
StringBuffer class constructors, 61, 63
StringBuffer class methods, 61-63
StringBuilder Class, 131-132
stringWidth, 288
StringWriter, 164
Struts, 522, 543, 545-546
StyleConstants, 412-413, 415-416
Submenus, 315-316, 336, 341, 350, 367, 549-550, 564-565
subpath, 620-621
Substance, 356
suspend, 141, 680, 684-685, 705-706, 711-713
switch statements, 17, 79, 95
Synchronization, 131, 676-677, 692-696, 699, 701-702, 718-724
Synchronizers, 696, 719, 722
Synchronizing, 673, 692-694, 696, 976
SynthLookAndFeel, 383-384
System Tray, 209-211

T

Tab Placement, 521, 528, 530, 532
Tabbed Panes, 507, 521-522, 524, 526-528, 530-532, 534, 536, 538, 540, 542, 544, 546, 548
Table Sorting, 444, 478
TableModel, 476-479, 481-482
TableModel Class, 477
TableModel interface, 476-478, 482
TableModelEvent, 483
TableRowSorter, 478-481
Task scheduling framework, 719
Telnet, 726, 740
Terminated, 675-676
ternary if-then-else, 77
Text Antialiasing, 209, 214
Text Fields, 171-174, 188, 192-193, 214-216, 218, 220, 222, 224, 226, 228, 230-232, 234, 236, 238, 240, 242, 244, 246, 250, 259, 276, 282, 344, 394-398, 400, 402-404, 406, 408, 410, 412, 414, 416, 418, 420, 422, 424, 426, 428, 430, 432, 434, 436, 438, 440, 442, 453-454, 490, 492, 507-509, 537-538, 542, 547-548
TextArea, 172, 248-252, 260, 271, 273, 328, 409, 414-416, 595, 597
TextEvent, 195

TextListener, 194
TextPane, 173
Thread class, 673-674, 678-679, 681-685, 688, 705, 707-708
Thread Priority, 673, 690, 705
three-tier application, 9
Three-Tier Models, 889
Throwable class, 132-133
thumb, 246, 265, 267-268, 445, 456-458
Tick Marks, 443, 456-458, 460-461, 488
Timed_Waiting, 675-676
Toggle, 316, 339-340, 354, 394-398, 400, 402, 404, 406, 408, 410, 412, 414, 416, 418, 420, 422, 424, 426, 428, 430-432, 434, 436, 438, 440, 442, 978
Tomcat, 9
Translucent, 579, 583-588
Translucent Windows, 579, 583-586, 588
Transmission Control Protocol (TCP), 726
tree.getLastSelectedPathComponent, 486-487
TreeMap, 767-769, 799-801
TreeMap class, 767, 799-801
TreeNode interface, 483-484
TreeSelectionListener, 486
TreeSet, 768-769, 778, 790-792
TreeSet class, 769, 790-791
try block, 109-110, 134, 136, 140
Two-Tier, 889-890

U

UIManager, 383, 386-389, 393, 511, 530, 537, 590, 595, 603
uk, 730
unboxing, 32, 67-69, 74
Unicast, 736
Uniform Resource Locator (URL), 726, 930, 956
Uniform Resource Names (URNs), 731
Uniform Translucency Window, 586
UPDATE annotation, 890
URI Instance, 732
URL class, 290, 411, 726, 731, 735, 749-752, 880
URLClassLoader, 735
URLConnection, 725-726, 734-735, 750-756
User Datagram Protocol (UDP), 726, 729

V

valueChanged, 260, 469-470, 472, 486-487
valueOf, 56-57, 88, 96, 113, 230, 232, 474, 486, 806-807, 809
Varargs methods, 32, 69, 71-72, 74
Vector, 35, 393, 465, 467, 477, 482-483, 492, 768-769, 810-813, 912, 977
VERTICAL_SCROLLBAR, 453
VERTICAL_SCROLLBAR_ALWAYS, 408-409, 411, 453-454
VERTICAL_SCROLLBAR_AS_NEEDED, 453
VERTICAL_SCROLLBAR_NEVER, 453
viewport, 353-354, 408, 410, 445, 447-452, 466
Viewports, 353, 442-448, 450, 452, 454, 456, 458, 460, 462, 464, 466, 468, 470, 472, 474, 476, 478, 480, 482, 484, 486, 488
virtual key code, 280
VSPACE, 188

W

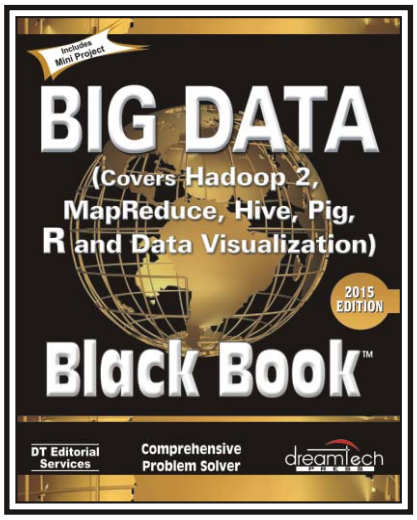
Waiting, 161, 276, 330, 352, 631, 638, 675-676, 681, 688, 695-698, 700-702, 704, 707, 716, 718, 721, 942
Warnings, 34, 830, 879, 881, 893, 896, 898
WeakHashMap, 769, 774
Web-based Enterprise, 7
WebLogic, 9-10, 886-888
Weblogic OCI Driver, 886
Weblogic RMI Driver, 887
WebSphere, 9
while loop, 17-18, 75, 83, 95-97, 99, 135, 483, 924
Whois, 725, 746, 766
widening conversion, 46
windowActivated, 321, 328-329
windowClosing, 146, 148, 174, 206-207, 321-323, 328-329, 347, 349, 414-416, 429, 517, 519, 589
windowDeactivated, 321, 328-329

WindowEvent, 146, 148, 174, 195, 206-207, 321-323, 325, 328-329, 347, 349, 375, 377, 414-416, 429, 517, 519, 589, 611
windowFocusListener, 323, 325, 328-329
WindowListener, 194, 206, 320-323, 325, 328-329
windowOpened, 321-322, 328-329
Windows look and feel, 388, 390
word-wrapping, 246
Wrapper Class, 32, 66-67
writeLock, 721-722
Writers, 616, 651, 721

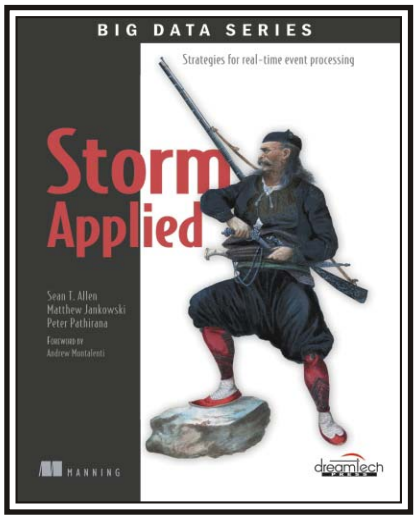
Z

Z-order, 184-185, 521-524
ZipEntry class, 839-840
ZipFile class, 838-839

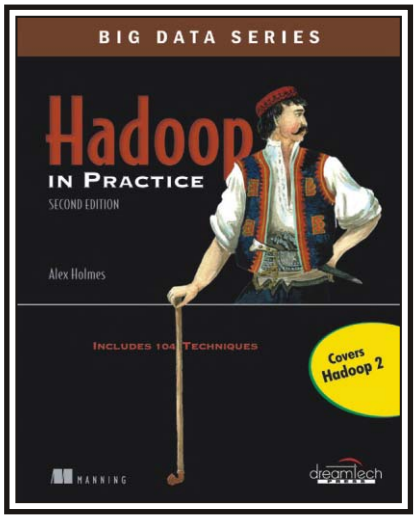
A WIDE RANGE OF BOOKS



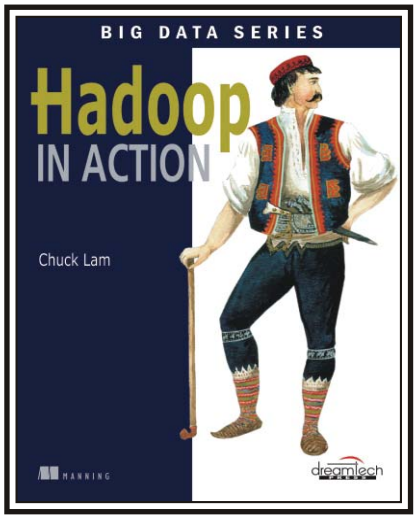
ISBN: 9789351197577



ISBN: 9789351197980

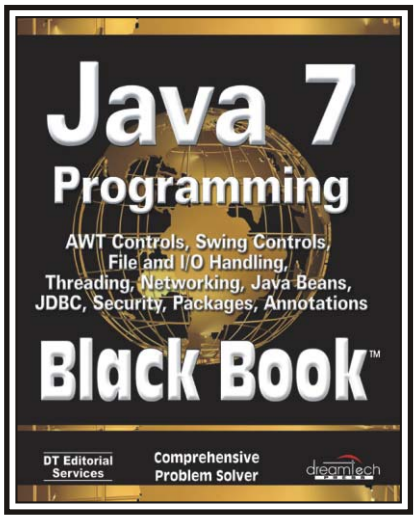


ISBN: 9789351197423

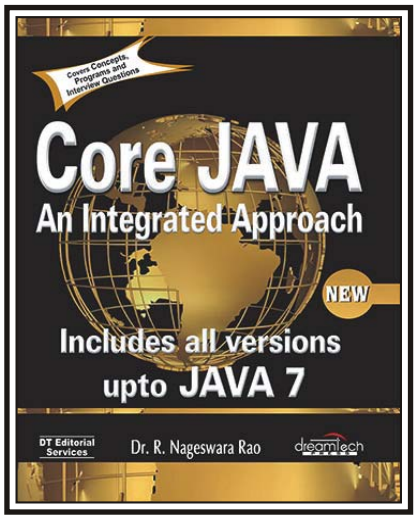


ISBN: 9788177228137

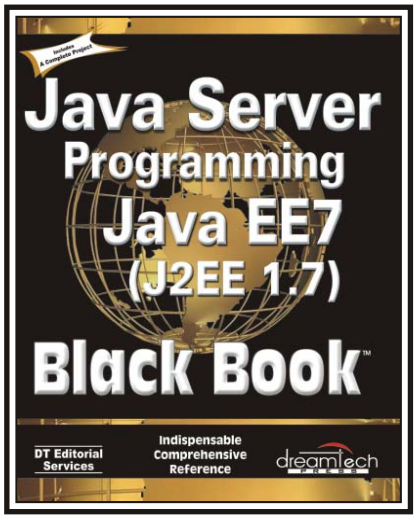
A WIDE RANGE OF BOOKS



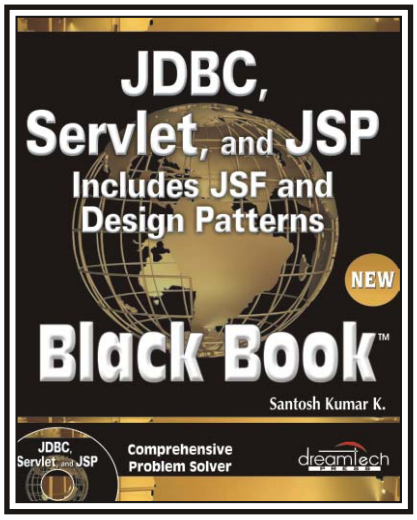
ISBN: 9789351190820



ISBN: 9788177228366

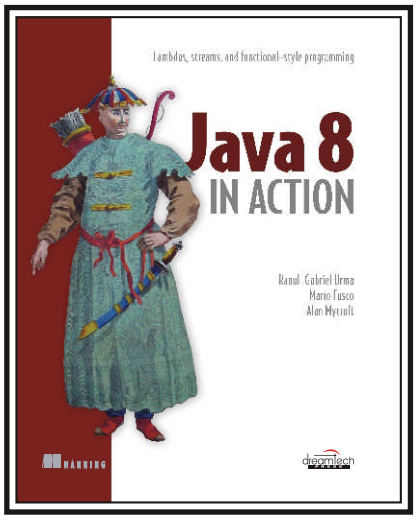


ISBN: 9789351194170

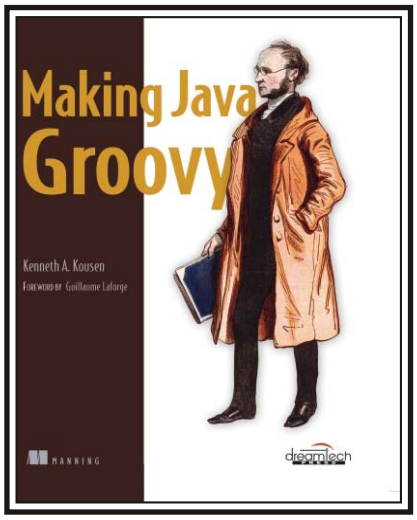


ISBN: 9788177228373

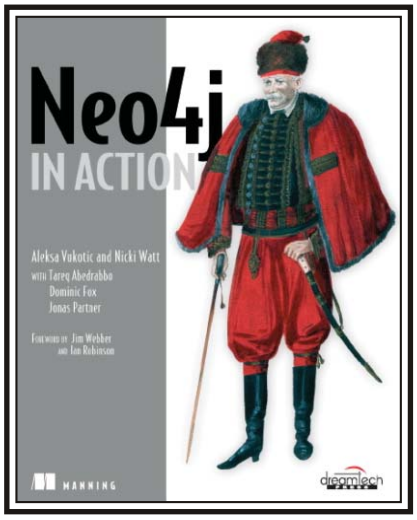
A WIDE RANGE OF BOOKS



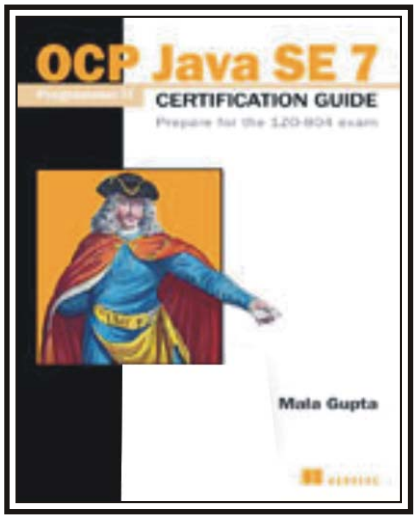
ISBN: 9789351197430



ISBN: 9789351194019

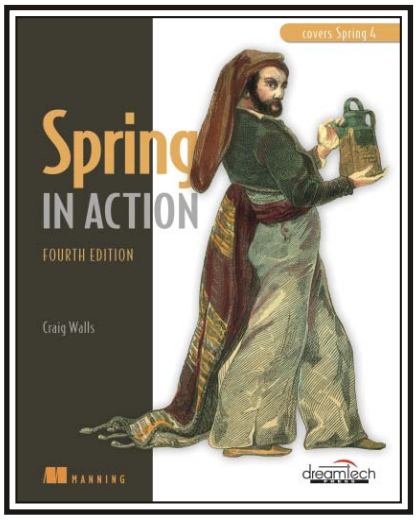


ISBN: 9789351197461

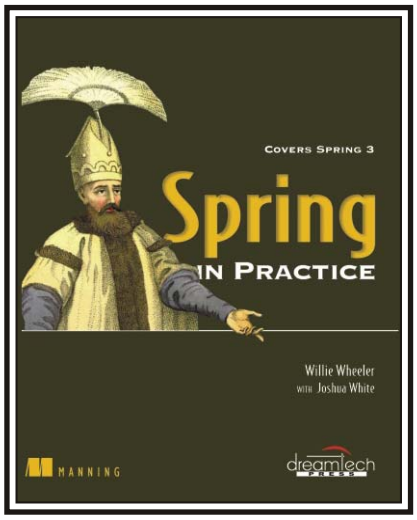


Coming Soon

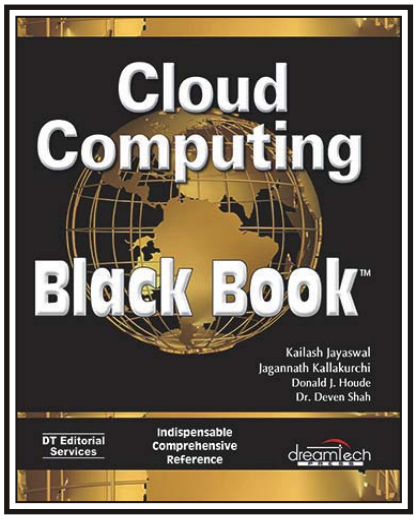
A WIDE RANGE OF BOOKS



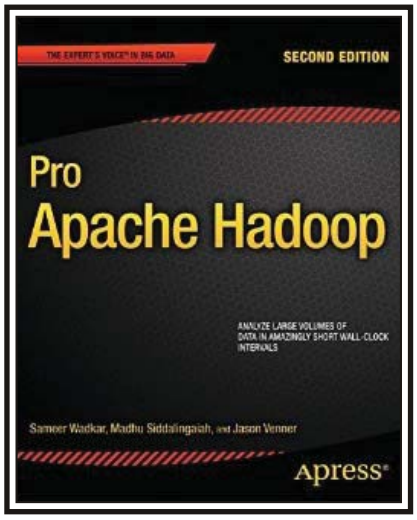
ISBN: 9789351197997



ISBN: 9789351192060



ISBN: 9789351194187



ISBN: 9781430248637

Java 8 Programming Black Book™

AWT Controls, Swing Controls, File and I/O Handling, Threading, Networking, Java Beans, JDBC, Security, Packages, Annotations Lambda expressions, NIO, Java DB, Java FX

Java 8 Black Book is a solid introductory reference that has been written from the experts' point of view and so includes hundreds of examples covering every feature of the subject. The book takes you in the direction of mastering the entire spectrum of Java 8—from generics to security enhancements, from new applet deployment enhancements to networking, from multiple threads to JavaBeans, from JDBC to file handling and much more. Designed to contain more on Java 8 than any other book on the topic, every aspect of this book is worth its price.

This book will help you to:

- Master the whole Java 8
- Comprehend Abstract Windowing Toolkit (AWT)
- Form multithreaded applications
- Handle advanced security issues
- Create both TCP clients and servers
- Conquer the Java Swing user interface
- Build packages, interfaces, and JAR files
- Generate JavaBeans
- Utilize Java Database Connectivity
- Employ networking and sockets on the Internet
- Specify through Generics
- Process Collections
- Create lists, choices, text areas, scroll bars, and scroll panes
- Create menus and toolbars

What the book has for you:

- Variables, Arrays, and Strings
- Operators, Conditionals, and Loops
- Object-Oriented Programming
- Inheritance, Inner Classes, and Interfaces
- AWT—Applets, Applications, and Event Handling
- AWT—Text Fields, Buttons, Checkboxes, Radio Buttons, and Layouts
- AWT—Lists, Choices, Text Areas, Scroll Bars, and Scroll Panes
- AWT—Graphics, Images, Text, and Fonts
- AWT—Windows, Menus, and Dialog Boxes
- Swing—Applets, Applications, and Pluggable Look and Feel
- Swing—Text Fields, Buttons, Toggle Buttons, Checkboxes, and Radio Buttons
- Swing—Viewports, Scrolling, Sliders, Lists, Tables, and Trees
- Swing—Combo Boxes, Progress Bars, Tooltips, Separators, and Choosers
- Swing—Layered Panes, Tabbed Panes, Split Panes, and Layouts
- Swing—Menus and Toolbars
- Swing—Windows, Desktop Panes, Inner Frames, and Dialog Boxes
- Streams—File and I/O Handling
- Multiple Threads
- Networking and Security with Java
- Collections
- Packages, Interfaces, JAR Files, and Annotations
- Java Beans
- JDBC

About the Author

DT Editorial Services has seized the market of computer books, bringing excellent content in software development to the fore. The team is committed to excellence—excellence in quality of content, excellence in the dedication of its authors and editors, excellence in the attention to detail, and excellence in understanding the needs of its readers.

Get Online Resources

Visit www.dreamtechpress.com/java-8-programming-black-book.html
Click the Companion Content tab.
Enter the Download code: JAVA8-7584-BB-1050
Click the Download button.

Also Available



CATEGORY:
Programming

USER LEVEL:
Intermediate to Advanced

978-93-5119-758-4



Published by:

dreamtech PRESS
19-A, Ansari Road, Daryaganj, New Delhi-110002
Tel.: 91-11-23284212, 23243075, 43551180, 41563722
Email: feedback@dreamtechpress.com
http://www.dreamtechpress.com