Java 2

(JDK 5 Edition)

AWT, Swing, Generics, XML, Sound, Animation, JDBC, Servlets, RMI, Threading, Sockets, Networking and Java Beans

Programming

New Edition

Black Book™

Comprehensive Problem Solver

Author Has Over 1 Million Books in Print!

Steven Holzner et al.

PARAGLYPH PRESS

dreamtech PRESS

This edition has been published by arrangement with the **Paraglyph Press, Inc., 2246 E. Myrtle Avenue, Phoenix Arizona 85202, USA** and published in India by **Dreamtech Press, 19-A, Ansari Road, Daryaganj, New Delhi-110002.**

Reprint Edition: 2008

Manufactured By: **Brijbasi Art Press Ltd., Okhla, Phase-II, New Delhi**

Contents at a Glance

1

47

Copyrighted image

105

149

209

idling 253

adio Buttons, and Layouts 305

s, and Scroll Panes 345

385

433

ig 479

529

565

629

CHAPTER 15
Swing: Applets, Applications, and Pluggable Look and Feel 673

CHAPTER 16
Swing: Text Fields, Buttons, Toggle Buttons, Checkboxes, and Radio Buttons 721

CHAPTER 17
Swing: Viewports, Scrolling, Sliders, Lists, Tables, and Trees 783

CHAPTER 18
Swing: Combo Boxes, Progress Bars, Tooltips, Separators, and Choosers 833

CHAPTER 19
Swing: Layered Panes, Tabbed Panes, Split Panes, and Layouts 873

CHAPTER 20
Swing: Menus and Toolbars 911

CHAPTER 21
Swing: Windows, Desktop Panes, Inner Frames, and Dialog Boxes 949

CHAPTER 22
Images and Animation 989

CHAPTER 23
Inclusion of Sound 1021

CHAPTER 24
Java and XML: Using the Document Object Model 1071

CHAPTER 25
Java and XML: Using the Simple API for XML 1113

CHAPTER 26
Collections 1147

CHAPTER 27
Creating Packages, Interfaces, JAR Files, and Annotations 1199

CHAPTER 28
Working with Java Beans 1221

CHAPTER 29
Talking to Database 1259

CHAPTER 30
JDBC in Action 1293

CHAPTER 31
Understanding RMI 1335

CHAPTER 32
Understanding Servlet Programming 1365

APPENDIX: A
Generic Types 1419

APPENDIX: B
Java Keywords 1457

☆☆☆

Table of Contents

Introduction xxxiii

CHAPTER 1
Essential Java 1

In Depth

All a Copyrighted image 3
Java Appears 4
All about Bytecodes 5
Java Security 5
Java Programs 6
Is it Java 2 (JDK 5 or JDK 1.5)? 8

Immediate Solutions

Getting and Installing Java 9
What about **CLASSPATH**? 10
What was new in Java 2, version 1.4 10
Novel Pipeline Architecture 10
Providing Hardware Acceleration for Offscreen Images 11
Supplying Pluggable Image I/O Framework 11
New Java™ Print Service 11
Assistance for float and double Image Types 11
Public Bidi Algorithm 11
Aiding Font Rasterizer for TrueType Hinting 12
Hinted Lucida Fonts 12
Help from OpenType Font Table 12
Numeric Shaping Support 12
Enhanced Complex-Layout Support in GlyphVector 12
Absolute Porter–Duff Support 12

Methods for Checking if Font has a Transform 13
Improved Equality Methods for FontRenderContext 13
What has been deprecated in Java 2, version 1.4? 13
What's new in Java 2, version 5 13
Caching Entire BufferedImages 13
Methods Related to Hardware Acceleration of Images 14
Hardware-Accelerated Rendering Using OpenGL 14
Solaris OpenGL Notes 15
Notes on Linux OpenGL 15
Microsoft Windows OpenGL-based Pipeline 15
Support for CUPS Printers— Solaris and Linux 15
Bicubic Interpolation Assistance 15
Generate Fonts from Files and Streams 15
Better Text Rendering Performance and Reliability 16
Rendering 16
Multilingual Text 16
What has been Deprecated in Java 2, version 1.5? 16
What is JSP? 16
What is J2EE? 17
What are Application Servers (Tomcat, Jrun, WebSphere, WebLogic, etc.)? 17
What is JSF? 19
What is J2ME? 19
What is Struts? 20
Subfolders created by JDK 20
Writing Code: Creating Code Files 21
Writing Code: Knowing Java's Reserved Words 21
Writing Code: Creating an Application 23
public class app 23
public static void main(String[] args) 24
System.out.println("Hello from Java!") 25
Compiling Code 26
Compiling Code: Using Command-Line Options 27
Cross-Compilation Options 28
Compiling Code: Checking for Deprecated Methods 28
Running Code 29
Running Code: Using Command-Line Options 31
Basic Skills: Commenting Your Code 33
Basic Skills: Import Java Packages and Classes 35
Basic Skills: Finding Java Classes with **CLASSPATH** 37
Creating Applets 39
Running Applets 40
Creating Windowed Applications 42
Running Windowed Applications 43

Designing Java Programs....43
Performance....44
Maintainability....44
Extensibility....45
Availability....45
Distributing Your Java Program....45

CHAPTER 2
Variables, Arrays, and Strings....47
In Depth
Variables....49
Data Typing....51
Arrays....51
Strings....53
Immediate Solutions
What Data Types Are Available?....55
Creating Integer Literals....56
Creating Floating-Point Literals....57
Creating Boolean Literals....58
Creating Character Literals....58
Creating String Literals....60
Declaring Integer Variables....60
Declaring Floating-Point Variables....61
Declaring Character Variables....62
Declaring Boolean Variables....63
Initializing Variables....64
Dynamic Initialisation....65
Converting between Data Types....65
Automatic Conversions....66
Casting to New Data Types....66
Declaring One-Dimensional Arrays....68
Creating One-Dimensional Arrays....69
Initializing One-Dimensional Arrays....69
Declaring Multi-dimensional Arrays....70
Creating Multi-dimensional Arrays....71
Initialising Multi-dimensional Arrays....72
Creating Irregular Multi-dimensional Arrays....73
Getting an Array's Length....74
General form of Static Import....75
Importing Static Members....76
The **String** Class....76
Creating Strings....80
Getting String Length....82
Concatenating Strings....83

Getting Characters and Substrings 83
Searching for and Replacing Strings 85
Changing Case in Strings 86
Formatting Numbers in Strings 86
The **StringBuffer** Class 87
Creating String Buffers 89
Getting and Setting String Buffer Lengths and Capacities 90
Setting Characters in String Buffers 91
Appending and Inserting Using String Buffers 91
Deleting Text in String Buffers 92
Replacing Text in String Buffers 92
Wrapper class 93
Autoboxing and Unboxing of Primitive Types 94
Varargs Fundamentals 98
Overloading Varargs Methods 100
Ambiguity in Varargs 102

CHAPTER 3
Operators, Conditionals, and Loops 105
In Depth
Operators 107
Conditionals 108
Loops 109
Immediate Solutions
Operator Precedence 111
Incrementing and Decrementing: **++** and **--** 112
Unary **Not**: ~ And **!** 114
Multiplication and Division: * and / 114
Modulus: **%** 115
Addition and Subtraction: **+** and **-** 115
Shift Operators: **>>**, **>>>**, and **<<** 116
Relational Operators: **>**, **>=**, **<**, **<=**, **==**, and **!=** 116
Bitwise and Bitwise Logical And, Xor, and Or: &, ^, and | 117
Logical **&&** and **||** 120
The **if-then-else** Operator: **?:** 121
Assignment Operators: **=** and **[operator]=** 122
Using the **Math** Class 124
Changes in the **Math** Class 125
Class StrictMath 125
Comparing Strings 126
The **if** Statement 127
The **else** Statement 128
Nested **if** Statements 129

The **if-else** Ladders 129
The **switch** Statement 130
The **while** Loop 132
The **do-while** Loop 134
The **for** Loop 136
The **for-each** Loop 139
Supporting **for-each** in Your Own Class 142
A (Poor) Solution 142
Significance of for-each 143
Nested Loops 144
Using the **break** Statement 145
Using the **continue** Statement 146

CHAPTER 4
Object-Oriented Programming 149
In Depth
Classes 152
Objects 152
Data Members 152
Methods 153
Inheritance 154
Exception Handling 154
Debugging 155
Immediate Solutions
Declaring and Creating Objects 156
Declaring and Defining Classes 159
Creating Instance Variables 161
Setting Variable Access 162
Creating Class Variables 163
Creating Methods 165
Setting Method Access 166
Passing Parameters to Methods 167
Command-Line Arguments Passed to **main** 169
Returning Values from Methods 170
Creating Class Methods 171
Creating Data Access Methods 172
Creating Constructors 173
Passing Parameters to Constructors 174
A Full Class Example 175
Understanding Variable Scope 176
Using Recursion 177
Garbage Collection and Memory Management 178
Avoiding Circular References 179
Garbage Collection and the **finalize** Method 180

Overloading Methods 181
Overloading Constructors 182
Passing Objects to Methods 183
Passing Arrays to Methods 185
Using the **this** Keyword 186
Returning Objects from Methods 187
Returning Arrays from Methods 188
New Classes Added to Java Lang 189
The Process Builder Class 189
The String Builder Class 190
Catching an Exception 192
Nesting **try** Statements 198
Using **Finally** Clause 199
Throwing Exceptions 201
Creating a Custom Exception 203
Debugging Java Programs 204

CHAPTER 5
Inheritance, Inner Classes, and Interfaces 209
In Depth
Why Inheritance? 211
Why Interfaces? 212
Why Inner Classes? 213
Immediate Solutions
Creating a Subclass 215
Access Specifiers and Inheritance 216
Calling Superclass Constructors 218
Creating Multilevel Inheritance 222
Handling Multilevel Constructors 224
Overriding Methods 225
Accessing Overridden Members 226
Using Superclass Variables with Subclassed Objects 228
Dynamic Method Dispatch (Runtime Polymorphism) 230
Creating Abstract Classes 232
Stopping Overriding with **final** 233
Stopping Inheritance with **final** 235
Creating Constants with **final** 236
Is-a vs. Has-a Relationships 236
The Java **Object** Class 237
Using Interfaces for Multiple Inheritance 240
New Interfaces added to Java Lang 242
The Readable Interface 243
The Appendable Interface 243
The Iterable Interface 244

Creating Iterable Objects 247
Creating Inner Classes 250
Creating Anonymous Inner Classes 251

CHAPTER 6
AWT: Applets, Applications, and Event Handling 253
In Depth
The Abstract Windowing Toolkit 255
Applets 256
Applications 257
Handling Events 258
Immediate Solutions
Using the Abstract Windowing Toolkit 259
Creating Applets 269
Using the <APPLET> HTML Tag 271
Handling Non-Java Browsers 274
Embedding <APPLET> Tags in Code 274
Using the init, start, stop, destroy, paint, and update Methods 275
Drawing Graphics in Applets 276
Reading Parameters in Applets 277
Using Java Consoles in Browsers 277
Adding Controls to Applets: Text Fields 278
Adding Controls to Applets: Buttons 280
Handling Events 281
Standard Event Handling 281
Using Delegated Classes 285
Using Action Commands 288
Handling Events the Old Way 289
Extending Components 289
Using Adapter Classes 291
Using Anonymous Inner Adapter Classes 293
Creating Windowed Applications 294
Exiting an Application When Its Window Is Closed 299
Applications You Can Run as Applets 300
Setting Applet Security Policies 301

CHAPTER 7
AWT: Text Fields, Buttons, Checkboxes, Radio Buttons, and Layouts 305
In Depth
Text Fields 307
Buttons 307
Checkboxes 307
Radio Buttons 307
Layouts 308

Immediate Solutions
Using Text Fields 309
Using Labels 312
Using Buttons 313
Using Checkboxes 317
Using Radio Buttons 321
Flow Layouts 323
Grid Layouts 326
Using Panels 329
Border Layouts 331
Card Layouts 333
Grid Bag Layouts 336
Using Insets and Padding 342
Creating Your Own Layout Manager 344

CHAPTER 8
AWT: Lists, Choices, Text Areas, Scrollbars, and Scroll Panes 345
In Depth
Lists 347
Choices 347
Text Areas 347
Scrollbars 348
Scroll Panes 348
Immediate Solutions
Using Text Areas 349
Replacing Text in Text Areas 352
Searching for and Selecting Text in Text Areas 354
Using Lists 356
Using Multiple-Selection Lists 360
Using Choice Controls 366
Using Scrollbars 372
Scrollbars and Border Layouts 377
Using Scroll Panes 380

CHAPTER 9
AWT: Graphics, Images, Text, and Fonts 385
In Depth
Graphics 387
Images 387
Text and Fonts 387
The Keyboard and Mouse 387
Immediate Solutions
Using the Mouse 388

Using the Keyboard 391
Using Fonts 395
Using Images 402
Resizing Images 405
Drawing Graphics 407
Drawing Lines 412
Drawing Ovals 413
Drawing Rectangles 413
Drawing Rounded Rectangles 414
Drawing Freehand 414
Drawing Arcs 415
Drawing Polygons 415
Setting Drawing Modes 415
Selecting Colors 415
Using Canvases 418
Using the **ImageObserver** Interface 420
Using the **MediaTracker** Class 421
Working Pixel by Pixel: 424
The **PixelGrabber** and 424
MemoryImageSource Classes 424
Brightening Images 427
Converting Images to Grayscale 429
Embossing Images 430

CHAPTER 10
AWT: Windows, Menus, and Dialog Boxes 433
In Depth
Windows 435
Menus 435
Dialog Boxes 436
Immediate Solutions
Creating Frame Windows 437
Showing and Hiding Windows 438
Handling Window Events 441
Automatically Hiding Windows upon Closing 443
Using the **Windows** Class 444
Creating Menus 451
Creating a **MenuBar** Object 453
Creating **Menu** Objects 455
Creating **MenuItem** Objects 456
Handling Menu Events 458
More Menu Options 460
Adding Menu Separators 462

Disabling Menu Items 463
Adding Checkboxes to Menus 464
Creating Submenus 466
Pop-Up Menus 468
Dialog Boxes 471
File Dialog Boxes 475

CHAPTER 11
Working with Streams: File and I/O Handling 479
In Depth
Streams, Readers, and Writers 481
Immediate Solutions
Using the **File** Class 482
Working with **InputStream** 486
Working with OutputStream 486
Working with FileInputStream 487
Working with FileOutputStream 489
Working with ByteArrayInputStream 491
Working with ByteArrayOutputStream 493
Working with BufferedInputStream 495
Working with BufferedOutputStream 497
Working with RandomAccessFile 498
Working with **Reader** 500
Working with **Writer** 501
Keyboard Input: Working with **InputStreamReader** 502
Working with **OutputStreamWriter** 503
Working with **FileReader** 503
Working with **FileWriter** 504
Working with **CharArrayReader** 505
Working with **CharArrayWriter** 507
Working with **BufferedReader** 509
Working with **BufferedWriter** 511
Working with **PushbackReader** 512
Working with **StreamTokenizer** 512
Working with **Serialization** 514
Working with the Clipboard 519
Working with the Printer 519
Printing with 1.5 formatter 520
System.out.printf 520
String.format 521
Dates 521
java.util.Formatter 522
Scanning Input with the 1.5 scanner class 523

Working with 524
NIO 524
Essentials in NIO 525
Buffers 525
Channels 526
Charsets and Selectors 528
Prospects of NIO 528

CHAPTER 12
Filing and Printing Documents 529
In Depth
Serializable 531
Externalizable 531
Printing in Java 531
Job Control 531
Imaging 532
Printable Jobs 532
Pageable Jobs 532
Immediate Solutions
Serializing the Sketch 533
Assigning a Document Name 533
Recording the Changes in Sketch 534
Creating a Sketch 535
Implementing the Serial Interface 547
Serializing the List of Elements 548
Serializing Lines 548
Serializing Rectangles 549
Serializing Circles 550
Serializing Curves 550
Supporting the File Menu 553
Printing in JAVA 554
Paper Class 555
Page Format Class 556
Printer Job Class 557
Book Class 558
Pageable Interface 559
PrinterGraphics Interface 559
Creating and Using PrinterJob Objects 560
Displaying a Print Dialog 560
Getting Started with Printing 561
Printing Pages 562
To Format a Page 563

CHAPTER 13
Working with Multiple Threads 565
In Depth
Using Threads in Java 567
Immediate Solutions
Getting the Main Thread 570
Naming a Thread 570
Pausing a Thread 571
Creating a Thread with the **Runnable** Interface 573
Creating a Thread with the **Thread** Class 576
Creating Multiple Threads 580
Waiting for (Joining) Threads 583
Checking Whether a Thread Is Alive 584
Setting Thread Priority and Stopping Threads 586
Why Use Synchronization? 587
Synchronizing Code Blocks 589
Synchronizing Methods 590
Synchronizing classes added in J2SE 5.0 592
Semaphores 593
Mutex 597
Cyclicbarrier 597
Communicating between Threads 603
Suspending and Resuming Threads 605
Creating Graphics Animation with Threads 609
Eliminating Flicker in Graphics Animation 612
Suspending and Resuming Graphics Animation 613
Double Buffering 615
Simplifying Producer-Consumer with 1.5 Queue Interface 618
Concurrent Programming with J2SE 5.0 621
Limitations of Synchronization in J2SE 1.4.x 622
Simplifying Servers using the Concurrency Utilities 622
Overview of the Concurrency Utilities 623
java.util.concurrent Package 624
java.util.concurrent.locks Package 626
Reader/Writer Locks 626
java.util.concurrent.atomic Package 627

CHAPTER 14
Networking with Java 629
In Depth
Basics of Networking 631
Sockets in Java 632
Client-Server in Networking 632

Proxy Servers 633
Internet Addressing 633
Domain Naming Service (DNS) 634
Inet4Addresses and Inet6Addresses 634
The URL Class 635
The URI Class 635
URI syntax and components 636
TCP/IP and Datagram 637
What is Bart? 637

Immediate Solutions

Java Net API's 639
The Networking Interfaces and Classes 639
InetAddresses 641
IP address scope 641
Host Name Resolution 641
InetAddress Caching 642
Factory Methods 644
Instance Method 645
Creating and Using Sockets 646
Creating TCP Clients and Servers 649
TCP/IP Client Sockets 651
A **Whois** Example 653
TCP/IP Server Sockets 654
Submitting an HTML Form from Java 656
Handling **URL** 658
Using **URLConnection** Objects 660
Working with Datagrams 665
DatagramPacket 666
Datagrams Server and Client 667
Working with Bart 669

CHAPTER 15
Swing: Applets, Applications, and Pluggable Look and Feel 673

In Depth

The Java Foundation Classes 675
Swing 676
Heavyweight vs. Lightweight Components 678
Swing Features 679
Graphics Programming Using Panes 679
Model View Controller Architecture 680

Immediate Solutions

Working with Swing 681
Preparing to Create a Swing Applet 687

Understanding Root Panes 689
Understanding Layered Panes 691
Understanding Content Panes 693
Changes in Working with Content Panes 694
Creating a Swing Applet 697
Painting in Swing vs. AWT 697
Displaying Controls in Swing vs. AWT 697
Using the JPanel Class 697
Creating a Swing Application 700
Closing **JFrame** Windows 703
Selecting Component Borders 706
Using Insets 708
The Synth skinnable look and feel 709
Setting the Pluggable Look and Feel 713
Setting the Pluggable Look and Feel for Components 718

CHAPTER 16
Swing: Text Fields, Buttons, Toggle Buttons, Checkboxes, and Radio Buttons 721

In Depth

Labels and Text Fields 723
Password Field 723
Text Area 724
Editor Pane 724
Text Pane 724
Buttons 724
Toggle Buttons 724
Checkboxes and Radio Buttons 724

Immediate Solutions

Using Labels 725
Using Image Icons 728
Using Images in Labels 729
Using Text Fields 731
Setting Text Field Alignment 733
Creating Password Field 734
Creating Text Areas 736
Customizing Text Areas 738
Creating Editor Panes 738
Using HTML in Editor panes 741
Using RTF files in Editor panes 741
Creating Text panes 742
Inserting Images and Controls into Text Panes 742
Setting Text Pane Text Attributes 744

Working with Sound in Applets 750
Working with Sound in Application 751
Abstract Button: The Foundation of Swing Buttons 751
Using Buttons 755
Displaying Images in Buttons 759
Using Rollover and Disabled Images 762
Default Buttons and Mnemonics 763
Using Toggle Buttons 767
Creating Toggle Button Groups 769
Using Checkboxes 770
Using Radio Buttons 775
Using Checkbox and Radio Button Images 780
Getting and Setting the State of Checkboxes and Radio buttons 781

CHAPTER 17
Swing: Viewports, Scrolling, Sliders, Lists, Tables, and Trees 783
In Depth
Viewports 785
Scroll Panes 785
Sliders 785
Scrollbars 785
Lists 785
Tables 786
Trees 786
Immediate Solutions
Handling Viewports 787
Creating Scroll Panes 791
Creating Scroll Pane Headers and Borders 796
Scrolling Images 797
Creating Sliders 798
Filling a Slider 802
Painting Slider Tick Marks 803
Painting Slider Labels 804
Setting the Slider Extent 804
Creating Scrollbars 806
Creating Lists 810
Handling Multiple List Selections 815
List Selection Modes 815
Displaying Images in Lists 817
Creating a Custom List Model 818
Creating a Custom List Cell Renderer 818
Handling Double Clicks in Lists 819
Creating a Table 821
Adding Rows and Columns to Tables at Runtime 823

Creating Trees 824
Adding Data to Trees 826
Handling Tree Events 829
Editing 831

CHAPTER 18
Swing: Combo Boxes, Progress Bars, Tooltips, Separators, and Choosers 833
In Depth
Combo Boxes 835
Progress Bars 835
Choosers 835
Tooltips 835
Separators 836
Immediate Solutions
Creating Combo Boxes 837
Handling Combo Box Selection Events 842
Creating Editable Combo Boxes 844
Adding Images to Combo Boxes 845
Creating a Combo Box Model 847
Creating a Combo Box Custom Renderer 847
Creating Progress Bars 848
Updating Progress Bars 852
Handling Progress Bar Events 853
Creating Tooltips 854
Creating Separators 856
Resizing Separators Automatically 859
Creating Color Choosers 860
Creating File Choosers 863
Creating File Chooser Filters 869

CHAPTER 19
Swing: Layered Panes, Tabbed Panes, Split Panes, and Layouts 873
In Depth
Layered Panes 875
Tabbed Panes 875
Split Panes 875
Layouts 875
Immediate Solutions
Understanding Swing Components and Z-order 877
Making Swing Components Transparent 879
Using Layered Panes 880
Creating Tabbed Panes 883

Specifying Tab Placement in Tabbed Panes 889
Using Split Panes 893
Making Split Panes One-Touch Expandable 898
Setting Split Pane Orientation 899
Setting Split Pane Divider Size 900
Using the Box Layout Manager 902
Using the **Box** Class 904
Using the Overlay Layout Manager 908

CHAPTER 20
Swing: Menus and Toolbars 911
In Depth
Menus 913
Toolbars 913
Immediate Solutions
Creating a Menu Bar 914
Creating a Menu 916
Creating a Menu Item 918
Creating a Basic Menu System 921
Adding Images to Menus 924
Creating Checkbox Menu Items 926
Creating Radio Button Menus 928
Creating Submenus 931
Creating Menu Accelerators and Mnemonics 932
Enabling/Disabling Menu Items and Changing Captions at Runtime 935
Adding and Removing Menu Items at Runtime 937
Adding Buttons and Other Controls to Menus 938
Creating Pop-Up Menus 940
Creating Toolbars 943
Adding Combo Boxes and Other Controls to Toolbars 947

CHAPTER 21
Swing: Windows, Desktop Panes, Inner Frames, and Dialog Boxes 949
In Depth
Windows 951
Dialog Boxes 951
Immediate Solutions
Creating a Window 952
Creating a Frame Window 955
Creating a Desktop Pane 956
Creating Internal Frames 958
Using **JOptionPane** to Create Dialog Boxes 968
Creating Option Pane Confirmation Dialog Boxes 974

Creating Option Pane Message Dialog Boxes 975
Creating Option Pane Text Field Input Dialog Boxes 978
Creating Option Pane Combo Box Input Dialog Boxes 979
Creating Option Pane Internal Frame Dialog Boxes 981
Creating Dialog Boxes with **JDialog** 982
Getting Input from Dialog Boxes Created with **JDialog** 986

CHAPTER 22
Images and Animation 989
In Depth
Using Images 991
Different Formats of Images 991
Using ImageIcon class 991
Understanding java.awt.geom Package 992
Immediate Solutions
Using **ImageIcon** Class 993
Creating an Image 993
Loading an Image 993
Displaying an Image 994
Drawing an Image Object 995
Loading an **Image** Object 999
ImageObserver 999
MediaTracker 1002
ImageIcon 1004
Using **ImageIcon** in an Applet 1004
Getting Images in Java Applets 1005
Drawing Images 1006
Image Observers 1009
Animation 1011
The Classes of geom 1011
Creating a Game using Class 1013
Collecting Your Images 1013
Organizing and Loading the Images in Your Applet 1013
Animating the Images 1014
Finishing Up 1017

CHAPTER 23
Inclusion of Sound 1021
In Depth
Simple Sound Output 1023
Sound API Basics 1024
What Is Sampled Audio? 1024
Understanding MIDI 1025

What Is MIDI? 1025
MIDI Configurations 1025
Service Provider Interfaces 1026

Immediate Solutions

Playing Sound 1028
Using a Clip 1029
Setting up the Clip for Playback 1029
Starting and Stopping Playback 1029
Playing Sounds with **javax.sound** 1030
Using a **SourceDataLine** 1035
Setting up the SourceDataLine for Playback 1035
Starting and Stopping Playback 1036
Monitoring a Line's Status 1038
Synchronizing Playback on Multiple Lines 1038
Processing the Outgoing Audio 1039
Streaming Sounds with javax.sound 1039
Sound Recording 1041
Setting up a TargetDataLine 1042
Reading the Data from the TargetDataLine 1042
Monitoring the Line's Status 1044
Processing the Incoming Audio 1044
Sound Events 1044
Introduction to Controls 1044
Getting a Line that has the Desired Controls 1046
Getting the Controls from the Line 1046
Using a Control to Change the Audio Signal 1047
Controlling a Line's Mute State 1047
Changing a Line's Volume 1048
Selecting among Various Reverberation Presets 1048
Manipulating the Audio Data Directly 1048
Reading Sound Files 1050
Writing Sound Files 1052
Converting File and Data Formats 1057
Converting from One File Format to Another 1057
Converting Audio between Different Data Formats 1059
What Conversions are available 1061
Working with MIDI 1061
A MIDI Refresher: Wires and Files 1063
Streaming Data in the MIDI Wire Protocol 1064
Sequenced Data in Standard MIDI Files 1064
The Java Sound API's Representation of MIDI Data 1065
MIDI Messages 1065
MIDI Events 1065

Sequences and Tracks 1065
The Java Sound API's Representation of MIDI Devices 1066
The MidiDevice Interface 1066
Transmitters and Receivers 1066
Sequencers 1066
Synthesizers 1067

CHAPTER 24
Java and XML: Using the Document Object Model 1071
In Depth
Document Object Model 1073
Writing XML Documents 1073
Valid and Well-Formed XML Documents 1077
Using XML for Java 1079
Using Java's JAXP 1082
Immediate Solutions
DOM Interfaces and DOM Implementations 1083
XML for Java: Creating a DOMParser Object 1083
Parsing an XML Document 1084
Getting the Parsed Document 1092
Displaying an XML Document 1095
Handling Document Nodes 1097
Handling Element Nodes 1098
Handling Attributes 1098
Handling Child Elements 1100
Handling Text Nodes 1101
Handling XML Processing Instruction Nodes 1102
Closing Elements 1103
Running the XML4JParser Application 1104
Navigating in XML Documents 1106
Creating a JAXP Application 1109

CHAPTER 25
Java and XML: Using the Simple API for XML 1113
In Depth
Using the SAX Parser 1115
Immediate Solutions
Creating a SAX Parser 1120
Registering an Event Handler with a SAX Parser 1130
Handling the Beginning of Documents 1132
Handling the Beginning of Elements 1133
Handling Attributes 1134
Handling Text Content 1136
Ignoring "Ignorable" White Space 1137

Handling the End of Elements 1138
Handling XML Processing Instructions 1138
Handling Errors and Warnings 1139
Running the XML4JSAXParser Application 1140
Creating a JAXP SAX Parser 1142

CHAPTER 26
Collections 1147
In Depth
The Collection Interfaces 1149
The Collection Classes 1150
The Map Interfaces 1150
The Map Classes 1150
Immediate Solutions
Using the **Collection** Interface 1151
Using the **List** Interface 1151
Using the **Set** Interface 1153
Using the **SortedSet** Interface 1153
Using the AbstractCollection Class 1154
Using the **AbstractList** Class 1155
Using the AbstractSequentialList Class 1156
Using the **ArrayList** Class 1157
Using the **LinkedList** Class 1159
Using the Generic Class 1161
Using the **HashSet** Class 1163
Using the **TreeSet** Class 1164
Using the **Comparator** Interface 1166
Using the **Iterator** Interface 1167
Using the **ListIterator** Interface 1168
Using the **AbstractMap** Class 1171
Using the **HashMap** Class 1172
Using the **TreeMap** Class 1176
Using the **Arrays** Class 1177
Enumeration Fundamentals 1180
The **Values()** and **Valueof()** Methods 1183
Java Enumeration as Class Type 1184
Enumeration Inheriting Enum 1186
Using the **Enumeration** Interface 1188
Using the **Vector** Class 1188
Using the **Stack** Class 1192
Using the **Dictionary** Class 1193
Using the **Hashtable** Class 1194
Using the **Properties** Class 1196

CHAPTER 27
Creating Packages, Interfaces, JAR Files, and Annotations 1199

In Depth

Creating Packages and Interfaces 1201
JAR Files 1201
New Java.Lang.Sub.Packages 1201
Java.Lang.Annotation 1201
Java.Lang.Instrument 1201
Java.Lang.Management 1201
Annotation Basics 1201
Other Built-In Annotations 1202
@Retention 1203
@Documented 1203
@Target 1203
@Inherited 1204
@Override 1204
@Deprecaded 1204
@SuppressWarnings 1204

Immediate Solutions

Creating a Package 1205
Creating Packages That Have Subpackages 1206
Creating an Interface 1206
Partially Implementing an Interface 1208
Extending Interface 1209
Using Interfaces for callbacks 1210
Creating a JAR File 1211
Getting the Contents of a JAR File 1212
Extracting Files from a JAR File 1212
Updating JAR Files 1213
Reading from JAR Files in Code 1213
Using **javac** to Get Classes from JAR Files 1216
Letting Applets Get Classes from JAR Files 1217
Marker Annotations 1218
Single Member Annotations 1219

CHAPTER 28
Working with Java Beans 1221

In Depth

What is Java Bean? 1224
Advantage of Java Beans 1225
Introspection 1226
Customizers 1226

Immediate Solutions
Understanding Java Beans 1227
Designing Programs Using Java Beans 1227
Creating Applets That Use Java Beans 1229
Creating a Java Bean 1231
Creating a Bean Manifest File 1233
Creating a Bean JAR File 1233
Using a New Bean 1234
Adding Controls to Beans 1234
Giving a Bean Properties 1235
Design Patterns for Properties 1242
Simple Properties 1243
Adding a Color Property to SimpleBean 1243
Indexed Properties 1245
Design Patterns for Events 1246
Methods and Design Patterns 1246
Using the BeanInfo Interface 1246
Feature Descriptors 1247
Creating Bound Properties 1247
Giving a Bean Methods 1248
Giving a Bean an Icon 1249
Creating a BeanInfo Class 1250
Bound and Constrained Properties 1252
Persistence 1254
The Java Beans API 1254
Events basics 1257
Using Java Beans Conventions 1258
Remote Notification and Distributed Notification 1258
Using Beans with JSP pages 1258

CHAPTER 29
Talking to Database 1259
In Depth
What does JDBC Do? 1261
JDBC Versus ODBC and other APIs 1262
Two-tier and Three-tier Models 1263
Introducing SQL 1264
Immediate Solutions
The JDBC package 1266
Relating JDBC to ODBC 1266
Types of JDBC drivers 1267
JavaSoft Framework 1268
Driver interface and **DriverManager** class 1269

Connection Interface 1269
Statement Interface 1269
PreparedStatement 1269
ResultSet 1269
ResultSetMetaData Interface 1277
DatabaseMetaData Interface 1277
The Essential JDBC program 1278
Import the java.sql package 1279
Load a JDBC Driver 1279
Establishing a Connection 1279
Create a Statement 1280
Execute the Statement 1280
Process the ResultSet 1280
Close the Statement 1281
Close the Connection 1281
Using a **PreparedStatement** Object 1282
When to Use a PreparedStatement Object 1282
Creating a PreparedStatement Object 1282
Supplying Values for PreparedStatement Parameters 1283
Using a Loop to Set Values 1284
Return Values for the Method executeUpdate 1284
The interactive SQL tool 1286
Using Tables 1287
Understanding the TableModel Interface 1287
Defining a **TableModel** 1288

CHAPTER 30
JDBC in Action 1293
In Depth

Data Types and JDBC 1295
Scrollable ResultSets 1296
Batch Updates 1296
Programmatic Updates 1297
Mapping Relational Data onto Java Objects 1298
Mapping SQL Types to Java Types 1298
Basic JDBC Types 1298
CHAR, VARCHAR, and LONGVARCHAR 1299
BINARY, VARBINARY, and LONGVARBINARY 1299
BIT 1300
TINYINT 1300
SMALLINT 1301
INTEGER 1301
BIGINT 1301
REAL 1301

DOUBLE 1301
FLOAT 1301
DECIMAL and NUMERIC 1302
DATE, TIME, and TIMESTAMP 1302
Advanced JDBC Data Types 1303
BLOB 1304
CLOB 1304
ARRAY 1304
DISTINCT 1304
STRUCT 1304
REF 1304
JAVA_OBJECT 1305
Examples of Mapping 1305
Simple SQL Statement 1306
SQL Statement with IN Parameters 1306
SQL Statement with INOUT Parameters 1306
Custom Mapping 1307
Immediate Solutions
The **Statement** and **PreparedStatement** interface 1309
Modifying Data 1313
Database Queries 1313
The **ResultSet** 1316
Retrieving Column Data for Specified Data Types 1318
Working with Null Values 1318
Working with Special Data Types 1318
Date 1318
Time 1319
Timestamp 1319
Big Numbers 1319
Working with Streams 1320
Calling Procedure 1320
Handling Errors 1322
SQLWarning 1322
PreparedStatements 1324
Batch Updates 1326
BLOBs and CLOBs 1329
Browsing a Database 1329
Load Driver 1329
Create Connection 1329
Create Statement 1330
Execute SQL Statement and Return ResultSet 1330
Iterate ResultSet 1330
A Dynamic SQL Select Program 1331

The next Method and Data Retrieval....1331
Load Driver and Get Database Connection....1332
Retrieve Table Name from Command Line Argument....1332
Build Select Statement....1332
Create Statement Object and Execute SQL Statement....1333
Create a ResultSetMetaData Object....1333
Traverse the ResultSet....1333

CHAPTER 31
Understanding RMI....1335
In Depth
Remote method invocation (RMI)....1337
Client/Server architecture....1338
Implementing RMI....1339
Limitation of RMI....1340
Immediate Solutions
A Model RMI Transaction....1342
Writing an RMI Server....1342
Designing a Remote Interface....1345
Implementing a Remote Interface....1347
Declaring the Remote Interfaces....1348
Defining the Constructor....1348
Providing Implementations for Each Remote Method....1349
Passing Objects in RMI....1349
Implementing the Server's **main** Method....1350
Creating and Installing a Security Manager....1350
Making the Remote Object Available to Clients....1350
Creating a Client Program....1352
Compiling and Running the Example....1355
Compiling the Example Programs....1355
Building a JAR File of Interface Classes....1356
Building the Server Classes....1356
Building the Client Classes....1357
Running the Example Programs....1358
A Note about Security....1358
Starting the Server....1359
Starting the Client....1360
Exporting with **UnicastRemote** Object....1361
Exporting **Activatable** Objects....1362

CHAPTER 32
Understanding Servlet Programming....1365
In Depth
What's new in Servlet 2.4?....1367

Upgraded support for HTTP, J2SE, and J2EE 1368
New ServletRequest Methods 1368
RequestDispatcher Changes 1368
Listeners 1369
Session Changes 1369
Miscellaneous Clarifications 1370
Deprecations 1370
Schema 1370
What you don't see 1371
Overview of Servlets 1371
Using Servlets 1372
Servlet API 1373
Features of the API 1373
What do Servlets Look Like? 1373
Servlet and Environment State 1374
Usage Modes 1374
Servlet Lifecycle 1374
When Are Servlets Loaded? 1375
Primary Servlet Methods 1375
Security Features 1376
HTML-Aware Servlets 1377
HTTP-Specific Servlets 1378
Performance Features 1378
Three-Tier Applications 1379
Web Publishing System 1380
Package javax.Servlet Description 1381
Generic Servlet Interfaces and Classes 1381
The javax.Servlet package 1381
The javax.Servlet Interfaces 1381
The javax.Servlet Exception Classes 1383
Servlet Configuration 1383
Creating an HTML File 1384
Creating a Servlet 1384
Creating a Web Application 1385
Deploying the Web Application 1386
Running the Web Application 1386
How the Application Works 1386
Client/Server Servlet Programming 1386
HttpServletRequest and HttpServletResponse 1387
HttpServletResponse 1387

Immediate Solutions

Servlet Life Cycle 1389
The init() Method 1390

The service() Method 1391
The doGet(), doPost(), and doXxx() Methods 1392
The SingleThreadModel Interface 1392
The destroy() Method 1393
Understanding Response and Request 1393
The Role of Form Data 1393
Reading Form Data from Servlets 1394
Response Headers 1396
Response Redirection 1397
Response Redirection Translation Issues 1400
Auto-Refresh/Wait Pages 1400
HttpServletRequest 1401
Headers 1401
Form Data and Parameters 1403
File Uploading 1406
Using a File Upload API 1413
Request Delegation and Request Scope 1415

APPENDIX: A
Generic Types 1419

APPENDIX: B
Java Keywords 1457

Introduction to Java

Thanks for buying the Java 2 Black Book. This book is designed to be as comprehensive and easily accessible as one book on Java can be. You're going to find as much Java crammed into this book as will fit between the covers.

Java is no ordinary programming language: It inspires devotion, passion, exaltation and eccentricity—not to mention exasperation and frustration. Hopefully what Java has to offer will prove as irresistible to you as it has to so many other programmers (in fact, Java programming is one of the most lucrative skills you can have today).

Java has been called "C++ for the Internet," and while there's some truth to that—especially with the Java/XML connection, which we cover in detail, that's proving so hot these days—the Internet is not the only place you'll find Java. More and more companies are using Java to build applications that have nothing to do with the Internet, but more and more to do with cross-platform reliability. I've seen many major corporations making the gradual shift from C++ to Java for in-house programming. Java's influence is spreading, and there's no sign of stopping it. With each new version, there's more power and more depth to work with.

If you're like me, you'll develop a taste for Java programming, because what you can do with this language is amazing. You'll see what I mean in page after page of this book.

Is This Book for You?

The Java 2 Black Book is designed to give you as much of the whole Java story as one book can hold. We'll see not only the full Java syntax—from declaring variables to advanced object-oriented issues—but also see Java in the real word. 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 will all be covered.

There are hundreds of topics covered in this book, and each of them will come with an example showing just how it works. This book is divided into separate, easily accessible topics each addressing a different programming issue. Here are just a few of those topics:

- The full Java 2 version 1.5 syntax
- Inheritance and inner classes
- Object-oriented programming
- The Abstract Windowing Toolkit (AWT)

- Generics
- Metadata
- Enumeration
- Autoboxing and Unboxing
- Java and XML
- Parsing XML with the DOM Model
- Parsing XML with the SAX Model
- Java Sound and Animation
- 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 the 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
- Packages, interfaces, and JAR files
- Java Server Pages (JSP)
- Reading applets from JAR files
- Security issues
- Java Database Connectivity (JDBC)
- Servlets
- RMI
- Exception handling
- Java collections
- Keyboard and mouse handling

That's just a partial list—there's a great deal more. This edition of the book also has special coverage for an especially hot topic—Java and XML. Together, Java and XML make a combination that just can't be beat, and it's turning up in more and more places. We'll get the full story in this book, with more coverage than some books dedicated to the subject.

How to Use This Book

In this book, I'll use Java 2, version 1.5. If you're not running at least this version, you might get some errors that seem mysterious as you run the code in this book, so pick up this version of Java (or later) on the Internet, as discussed in Chapter 1(currently, the URL is **http://java.sun.com/j2se/1.5/**).

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 plain - text format. See the topic "Writing Code: Creating Code Files" in Chapter 1 for more details.

Just about everything you need to use this book—besides an editor program—you can get from the Sun Java site, **http://java.sun.com**. The JDK has all you need to create standard Java applets and applications, and even has an applet viewer for displaying applets at work.

Besides the JDK, I'll also use the Beans Development Kit (BDK), and the Java Servlet Development Kit (JSDK) in this book, and you can get those from the Java site, too. We'll also take a look at Java Server Pages (JSP), so you might want to find a Web server that supports JSP. If you want to follow along with the database programming in this book, you'll need to create an ODBC data source on your machine. You'll find the database file that acts as this data source on the CD, along with all the code, images, and other files used in the book—not to mention a great number of other tools.

There's also one convention that I'll use in this book that you should be aware of. When a particular line of new code needs to be pointed out, I'll shade it this way:

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

Finally, Java comes with an immense amount of documentation—hundreds of books' worth, in fact. That documentation is stored in linked HTML pages, and you should have a Web browser to work with and view that documentation.

Other Resources

There are other Java resources that can be of assistance with Java. As mentioned, there are tens of thousands of pages of documentation that comes with Java it-self. There are also many, many Web pages out there on Java (a random Web search turns up a mere 10,268,200 pages mentioning Java; in fact, searching for "Java tutorial" alone turns up 11,614 pages). Here are some other useful resources:

- The Java home page is **http://java.sun.com/**.
- The Sun Java tutorial is at **http://java.sun.com/docs/books/tutorial/**.
- Sun's online technical support is at **http://developer.java.sun.com/developer/support/**.
- The Java 2 version 1.5 documentation itself is online at **http://java.sun.com/j2se/1.5/docs/index.html**.
- To get Java itself, go to **http://java.sun.com/j2se/1.5/**.

Among other topics, you can find tutorials on these topics at the Sun tutorial page: **http://java.sun.com/docs/books/tutorial/**:

- Collections
- Internationalization
- Servlets
- 2D Graphics
- Security in Java 2 version 1.5
- Sound
- Java Archive (JAR) Files
- Java Beans
- Java Database Connectivity (JDBC)
- Java Native Interface
- Remote Method Invocation (RMI)
- Reflection

There are also a number of Usenet groups for Java programmers, including:

- comp.lang.java.advocacy
- comp.lang.java.announce
- comp.lang.java.beans
- comp.lang.java.corba
- comp.lang.java.databases
- comp.lang.java.gui
- comp.lang.java.help
- comp.lang.java.machine
- comp.lang.java.programmer
- comp.lang.java.security
- comp.lang.java.softwwaretools

There's a lot of help available on Java, but I'm hoping you won't have to turn to any of it. This book, updated by editors at Cybermedia Services and Dreamtech Press, is designed to give you everything you need.

The Black Book Philosophy

Written by experienced professionals, Paraglyph Black Books provide *Immediate Solutions* to global programming and administrative challenges, helping you complete specific tasks, especially critical ones that are not well documented in other books. The Black Book's unique two-part chapter format—thorough technical overviews followed by practical immediate solutions—is structured to help you use your knowledge, solve problems, and quickly master complex technical is-sues to become an expert. By breaking down complex topics into easily manage-able components, this format helps you quickly find what you're looking for, with the diagrams and code you need to make it happen.

1

Essential Java

If you need an immediate solution to:	See page:
Getting and Installing Java	9
What about CLASSPATH?	10
What was new in Java 2, version 1.4	10
Novel Pipeline Architecture	10
Providing Hardware Acceleration for Offscreen Images	11
Supplying Pluggable Image I/O Framework	11
New Java™ Print Service	11
Assistance for float and double Image Types	11
Public Bidi Algorithm	11
Aiding Font Rasterizer for TrueType Hinting	12
Hinted Lucida Fonts	12
Help from OpenType Font Table	12
Numeric Shaping Support	12
Enhanced Complex-Layout Support in GlyphVector	12
Absolute Porter–Duff Support	12
Methods for Checking if Font has a Transform	13
Improved Equality Methods for FontRenderContext	13
What has been deprecated in Java 2, version 1.4?	13
What's new in Java 2, version 5	13
Caching Entire BufferedImages	13
Methods Related to Hardware Acceleration of Images	14
Hardware-Accelerated Rendering Using OpenGL	14
Solaris OpenGL Notes	15
Notes on Linux OpenGL	15
Microsoft Windows OpenGL-based Pipeline	15

If you need an immediate solution to:	See page:
Support for CUPS Printers— Solaris and Linux	15
Bicubic Interpolation Assistance	15
Generate Fonts from Files and Streams	15
Better Text Rendering Performance and Reliability	16
Rendering Multilingual Text	16
What has been Deprecated in Java 2, version 1.5?	16
What is JSP?	16
What is J2EE?	17
What are Application Servers (Tomcat, Jrun, WebSphere, WebLogic, etc.)?	17
What is JSF?	19
What is J2ME?	19
What is Struts?	20
Subfolders created by JDK	20
Writing Code: Creating Code Files	21
Writing Code: Knowing Java's Reserved Words	21
Writing Code: Creating an Application	23
public class app	23
public static void main(String[] args)	24
System.out.println("Hello from Java!");	25
Compiling Code	26
Compiling Code: Using Command-Line Options	27
Cross-Compilation Options	28
Compiling Code: Checking for Deprecated Methods	28
Running Code	29
Running Code: Using Command-Line Options	31
Basic Skills: Commenting Your Code	33
Basic Skills: Import Java Packages and Classes	35
Basic Skills: Finding Java Classes with CLASSPATH	37
Creating Applets	39
Running Applets	40
Creating Windowed Applications	42
Running Windowed Applications	43
Designing Java Programs	43
Performance	44
Maintainability	44
Extensibility	45
Availability	45
Distributing Your Java Program	45

In Depth

Welcome to our big book of Java programming. In this book, 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're like me and have a few programming packages, you'll enjoy working with Java more than others, and I hope that you'll choose Java as your choice of programming platform.

This first chapter covers the fundamental Java skills that you'll rely on in the coming chapters. In the next few chapters, you're going to see a great amount of Java syntax at work, but none of that is going to 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 to test out the Java syntax we'll develop.

In this chapter, we're going to work through the mechanics of creating Java programs—from installation issues to writing Java code; from making sure your Java program can find what it needs to display 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 Java code; this chapter is all about the rest of the process that makes the code run.

You might know much of the material in this chapter already; 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're 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. In this chapter, we will 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 with programmers, and it seemed that C was the perfect programming language combining the best elements of low-level assembly language and higher-level languages into a programming language that fits into computer architecture well and that programmers liked.

However, C also 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 code in the first line of a long program could interfere with 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 object-oriented programming, the motto is "divide and conquer". In other words, you can 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 to make 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 to make 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 have 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 helped 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++, and programmers found the MFC library a tremendous improvement over the old days. When 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 our 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 popularisation of what amounts to an immense new programming environment—the Internet. And that's what's 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 it was used internally at Sun.

The original idea for 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 popularised 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 modelled after Java). The original impetus for Oak was not what you'd call especially glamorous—Sun wanted to create a language it could use 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.

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 to let 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, Java programs are *compiled* into compact bytecodes, and it's these bytecodes that the JVM reads and interprets to run your program. When you download a Java applet on the Internet, you're 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 to work on those computers. Because your program is stored in a bytecode file, it will run on any computer on which JVM is installed.

Although Java programs were originally supposed to be *interpreted* by the JVM—that is, executed bytecode by bytecode—interpretation can be a slow process. For that reason, Java 2 introduces the Just In Time (JIT) compiler, which is built into the JVM. The JIT compiler actually reads your bytecodes in sections and compiles them interactively into machine language so the program can run faster (the whole Java program is not compiled at once because Java performs runtime checks on various sections of the code). From your perspective, this means your Java programs will run faster with the new JIT compiler.

Using bytecodes means that Java programs are very compact, which makes them ideal for downloading over the Internet. And there's another advantage of running such programs with the JVM rather than downloading full programs—Security.

Java Security

When Java executes a program, the JVM can strictly monitor what goes on, which makes it great for Internet applications. As you're 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. The 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. For example, 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.

As you can see, 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 robust (which means they're reliable and can handle errors well), they're often simple to write compared to C++, they're multithreaded (they can perform a number of tasks at the same time, which is useful when you want to continue doing other things while waiting for an entire data file to be downloaded, for example), and they offer high performance. 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.

Java is not only targeted at the Internet, of course; in fact, there are two types of Java programs—one for Internet use and one for local machine use.

Java Programs

Java programs come in two main types: applications and applets. (I'll use the term program in this book to refer to both applets and applications.) Applets are Java programs you can download and run in your Web browser, and they're what have made Java so popular. For example, you can see an applet at work in Figure 1.1, where the applet is running in Microsoft Internet Explorer and is displaying a greeting.

The major Web browsers have sometimes been slow to implement the most recent versions of Java, so in this book, I'll use the Sun applet viewer utility, which comes with Java, to look at applets. You'll find this utility (which is called applet viewer, with an extension as required by your system, such as applet viewer.exe in Windows) in the Java bin directory along with the other Java tools; we'll be using it in this and the next few chapters. You can see the applet viewer at work in Figure 1.2, displaying the same applet shown in Figure 1.1.

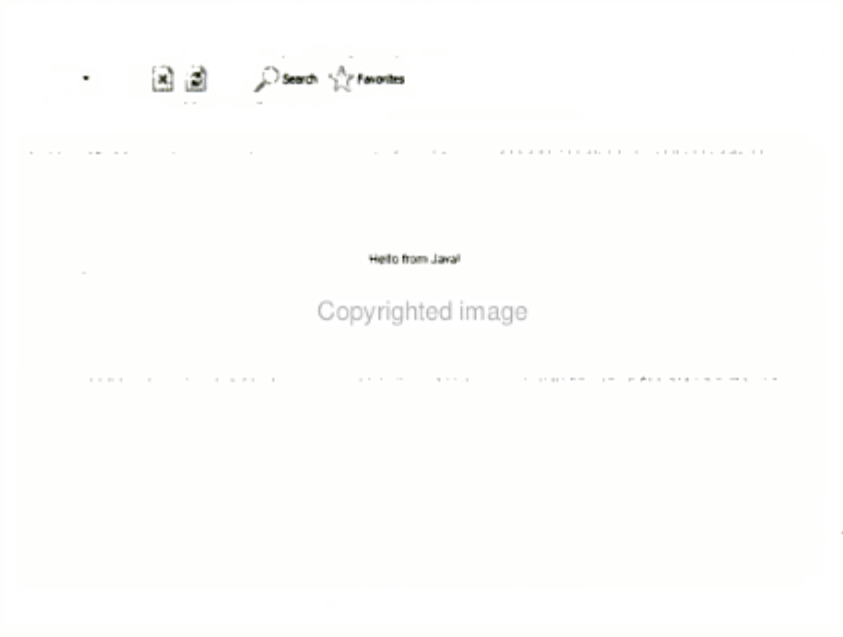

Figure 1.1 An applet displaying a greeting.

Figure 1.2 An applet at work in the Java applet viewer.

It is very important to realize that the major browsers may not be up to Java 2, version 1.5 by the time you read this, so some examples we develop here will be ahead of what those browsers can do. Does the fact that Web browser manufacturers are slow to upgrade to the latest Java version mean that you can't take advantage of what's new in version 1.5 in applets you use in those browsers? No. Sun has taken charge here and created a Java Plug-in for Netscape Navigator and Microsoft Internet Explorer that implement the latest Java version as a Netscape plug-in and Internet Explorer ActiveX control, respectively. This means that you can now run the latest Java applets in those browsers as long as you make some changes to the Web page the applet is in. You'll see all this in the chapter that formally introduces applets, Chapter 6, after we cover enough Java syntax to start creating applets.

So, keep in mind that the code we develop here may be ahead of the type of code that's supported in the browser you have—I get plenty of letters claiming that a certain example won't work in, say, Netscape Navigator 2.0 and asking why the code wasn't checked. Of course, the code is all checked; in fact, it's checked three times (including being checked by some very careful technical editors). The problem is that the browser doesn't support the current version of Java. The applet viewer will always be able to handle the most recent Java code. If your browser doesn't yet, take a look in Chapter 7 for information on how to get the free Sun browser plug-in.

Besides downloadable applets, Java also supports applications designed to be run on the local machine. Java applications work like other computer applications—you can install and run them on your computer. Because they're installed on your computer rather than just downloaded with a Web page, applications have more privileges than applets do, by default, such as the capability to read and write files.

The applications we'll be using in the next few chapters will be the simplest type of Java applications—the console applications. These are text-based applications you run from the command line (in Windows, this means from a DOS window), and they can read and display text. You'll see how such applications work in this chapter in detail. For example, say you have a Java application named app that prints out the message "Hello from Java!" (as the applet you've just seen does). In this case, the application will display that message in the console. You run the application by starting the Java Virtual Machine with the **java** command, passing it the name of the application you want to run. In Unix, that might looks like this, where "$" is the command prompt:

```
$java app
Hello from Java!
```

In Windows, it might look like this:

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

Figure 1.3 A Windowed Java Application.

Can Java applications be graphical? They certainly can, and in fact, it's safe to say that the majority of them are. In this case, the application is responsible for setting up its own window (which is done by the browser for applets). You can see a windowed Java application in Figure 1.3.

TIP: *I'll introduce windowed applications and at the same time introduce applets, in Chapter 7, where we start working with the Java Abstract Windowing Toolkit (AWT), which is how you create and manipulate windows with Java. We'll also take a quick look at creating windowed applications in this chapter, in case you just can't wait.*

Is it Java 2 (JDK 5 or JDK 1.5)?

There is one last area to take a look at before we begin—the actual Java version. The actual package we'll use is—the Java Software Development Kit (SDK) which includes the Java compiler, JVM, and other tools, you will need is now called *Java 2, version 1.5.*

Generally, one aspect of terminology causes confusion—the Java Software Development Kit has been referred to at various times as the SDK—Software Development Kit and as the JDK—Java Development Kit. So, if you see JDK, this generally means the same as SDK. Just for consistency, we'll use SDK to refer to any development kit in this book.

Here, we want to bring one more point into consideration that normally version 1.5 would have followed version 1.4, but it was decided to identify it as version 5.0 in recognition of the significance of the new feature that has been introduced by version 5.0 and the maturity of the product. Code module named in version 5.0 still uses the denotation 1.5.0 so we find folder names incorporating 1.5.0 rather than 5.0. So, don't let this confuse you.

That's enough overview and I believe that by this time you would have got a fair idea about Java and it's now time to start creating Java programs and seeing what goes into the process.

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://java.sun.com/j2se/1.5.0/** (*1.5.0* refers to the version of the SDK, and *j2se* stands for Java 2 Standard Edition).

After downloading the SDK, usually as one executable package that installs itself, follow the installation instructions on the java.sun.com site; for example, the Windows installation instructions are at **http://java.sun.com/j2se/1.5.0/install-windows.html**, and those for Solaris are at **http://java.sun.com/j2se/1.5.0/install-solaris.html.**

I'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 one that's designed to be as complete as possible. I've been writing about Java ever since it first came out, and it turns out that the actual installation instructions are often very volatile. Because these instructions changed, the instructions I provided in the previous books instantly became obsolete, triggering a landslide of calls and letters. 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 just involves 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, the bin directory is C:\jdk1.5\bin for SDK 2, version 1.5, so for Windows 95, 98, Millennium Edition, Windows XP, you simply add a line something like this (be sure to include any other directories you want to be in your path) to autoexec.bat:

```
SET PATH=C:\WINDOWS;C:\JDK1.5\BIN
```

TIP: *The Java Software Development Kit, was called the Java Development kit, JDK, in earlier Java versions. In fact, Sun often seems to be of two minds here—sometimes you'll see references to the SDK, and sometimes you'll see references to the JDK. This is one of those times—although the kit is now called the SDK, it's installed by default into a directory named \jdk1.5. Also, you must reboot your computer to make these changes take effect .*

In Windows 2000/Windows XP, you can follow these steps:

1. Open the Start menu and select Settings | Control Panel. Double-click the System icon.
2. In the System Properties dialog box, click the Advanced tab followed by the Environment Variables button.
3. Click the **PATH** variable.
4. Edit the **PATH** setting the way you want it and click OK.

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 2. 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 I'm 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 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, then you need to use the command line option to define the **CLASSPATH** temporarily.

Before moving ahead in this chapter, we need to know the features of Java 2, version 1.5 as against Java 2, version 1.4. Swing ahead for the details.

What was new in Java 2, version 1.4

"OK", the Novice Programmer (NP) says, "So, what was new in Java 2, version 1.4?" "Lots of things", you say. "Better get some coffee".

Here's an overview of what was new in Java 2, version 1.4.

Novel Pipeline Architecture

In the Java 2 SDK, versions 1.2 and 1.3, common operations on a **Graphics** object often invalidated the rendering data cached for this **Graphics** object. This invalidation interrupted the rendering process by causing continuous recreation of rendering information for the **Graphics** object, even for such simple and benign operations as create(), setColor(), and translate(). Because the rendering of Swing hierarchies relies heavily on these common operations, the invalidation and recreation of rendering data caused poor repaint performance for many Swing applications.

The new pipeline architecture reduces this performance overhead with several implementation changes that:

- Improves the way data is shared by the various rendering pipelines.
- Reduces the amount of code executed and garbage created when responding to changes in the rendering attributes.
- Improves the way that various graphics routines are chosen such as the routines that copy pixels from one format and location to another.

These changes are especially noticeable when the following calls are used frequently, as they are in Swing applications:

- getGraphics, Graphics.create() and Graphics.dispose()

- Graphics.setColor(), Graphics.translate
- Graphics.copyArea, especially when the source and destination regions overlap.

The runtime footprint should also be improved through better code sharing.d

Providing Hardware Acceleration for Offscreen Images

The SDK 1.4 provides access to hardware acceleration for offscreen images. This results in better performance of rendering to and copying from these images. The problem with hardware-accelerated images is that, on some platforms such as Microsoft Windows, their contents can be lost at any time due to some circumstances which are beyond the application's control. Therefore the new VolatileImage class allows you to create a hardware-accelerated offscreen image and manage the contents of that image.

Supplying Pluggable Image I/O Framework

The Java™ Image I/O API is a pluggable, extensible framework which allows reading and writing images of various formats and protocols. Support is given by the API through plug-ins, most of which will be written by third parties. An implementation meeting the requirements will only be required to provide a minimal set of plug-ins, principally for compatibility with previous versions of the Java SDK. An application using this API should be able to read and write images without knowing the image's storage format or the plug-in used to support the format.

New Java™ Print Service

This API, a product of JSR006, Unified Printing API, and will allow client applications to provide rich access to the capabilities of print services available including:

- printer browsing and selection;
- locating the capabilities of printers;
- selection of printers for a printer job;
- specification of a printer job.

Since all capabilities will be exposed through the API, server applications become first class citizens of this API. Server applications can be beneficiaries of the capabilities for spooling documents to print services, whereas in the past only graphics calls could be used to generate printer jobs from Java applications.

Assistance for float and double Image Types

In the previous version of SDK, the Java 2D API did not have the **DataBuffer** subclasses for float or double sample types. The Java Image I/O API needs these classes to read and write float and double image types.

The SDK 1.4 contains two new classes to provide float and double image type support—**DataBufferFloat** and **DataBufferDouble**. While the **DataBufferFloat** class wraps float arrays of pixels, the **DataBufferDouble** class wraps double arrays of pixels.

The existing **ComponentColorModel** and **ComponentSampleModel** class implementations have also been updated to support signed short, float, and double data.

Public Bidi Algorithm

First the Unicode Bi-directional Algorithm analyses text using the Unicode character properties and then determines the direction of runs of the text. The algorithm is necessary to properly display bi-directional text, such as Hebrew and Arabic text, in the correct order.

Though the current implementation is all written in the Java programming language, the SDK 1.4 will include efficient access from native font code so that Hebrew and Arabic text can be rendered more efficiently. The SDK 1.4 will provide access to the native code through the Java Native Interface.

The new public **Bidi** class implements the Unicode 3.0 Bidi Algorithm and allows access to information on the bi-directional reordering of the text. This is done so that the mixed, bi-directional text is properly displayed.

Aiding Font Rasterizer for TrueType Hinting

Prior to this release, the T2K font rasterizer used by Java 2D did not support font hinting for TrueType fonts. Thus, the TrueType fonts did not always display with a consistent, attractive appearance. In this release, the T2K rasterizer has been modified to use the hints stored in the TrueType fonts.

By adding this functionality to the T2K rasterizer, dependency on native rasterizers has been greatly reduced. Reducing this dependency results in:

- better portability because hinting of TrueType fonts is performed by the cross-platform T2K rasterizer, not the native rasterizer; and
- additional consistent metrics display of TrueType fonts because the same rasterizer is being used for on-screen and off-screen drawing.

Hinted Lucida Fonts

For the SDK 1.4, the Lucida fonts that are in the Java 2 SDK will be upgraded to contain hints, which will give the Java 2 SDK higher quality fonts that could be used in place of existing fonts or if no other fonts are available. Adding hints to the Lucida fonts also allows the new cross-platform rasterizer to hint that the Lucida fonts contained in the SDK causes these fonts to be displayed in a more consistent and attractive manner.

Help from OpenType Font Table

The SDK 1.4 includes a new architecture for providing general OpenType font support. This new architecture provides international character support for contextual scripts like Thai, Indic, Arabic, and Hebrew. It also provides enhanced typographical support for Roman languages.

Numeric Shaping Support

Now a days, when Java 2D renders numerals surrounded by Arabic text, the numerals have Arabic (roman) shapes, which are the commonly expected numeral shapes in most Western countries. However, people in a Hindi locale expect to see Hindi shapes. A new attribute, **TextAttribute.NUMERIC_SHAPING**, and a new class, **NumericShaper**, enable you to shape ASCII digits to other Unicode decimal ranges.

Enhanced Complex-Layout Support in GlyphVector

Before this release, clients could not access glyph-to-character mapping information from **GlyphVector**. Clients can use this information to find out which glyphs in the **GlyphVector** correspond to which characters. This release also defines new methods to get the precise bounds of the **GlyphVector** and of individual glyphs within a **GlyphVector**.

Absolute Porter–Duff Support

According to modes or rules established by Porter and Duff, this **AlphaComposite** class provides alpha blending capabilities. Out of the 12 rules which Porter and Duff had identified, the **AlphaComposite** API for the SDK, version 1.3 defines and implements only 8 of them. The rest 4 have been defined and

implemented in version 1.4. The **AlphaComposite** implements the remaining 4 Porter-Duff rules which are as follows:

- B (Dst)
- A atop B (SrcAtop)
- B atop A (DstAtop)
- A xor B (Xor)

Methods for Checking if Font has a Transform

Version 1.4, includes two new methods which allow you to check if a **Font** object's transform is an identity transform without creating a new **AffineTransform** that is:

- **Java.awt.Font.isTransformed:**
  It returns true if this **Font** object has a non-identity **AffineTransform** attribute.
- **Java.awt.font.TransformAttribute.isIdentity:**
  It returns true if the wrapped transform is an identity transform

Improved Equality Methods for FontRenderContext

A **FontRenderContext** object encapsulates the state about the graphics context and is used by **GlyphVector** and **TextLayout**. Three new methods in **FontRenderContext** allow you to compare the **FontRenderContext** in the **GlyphVector** against the one in the graphics context into which the **GlyphVector** draws:

- equals(FontRenderContext)
- equals(Object)
- hashCode()

These equals' methods also have performance benefits because a client does not have to create an **AffineTransform** to perform an equality test.

What has been deprecated in Java 2, version 1.4?

"Well", says the Novice Programmer, "I've installed Java 2, version 1.4 now, and I'm just about ready to go—but first, what's been deprecated?" "There is very little deprecation in the version 1.4", you say.

In fact, a few classes, fields, interfaces, and methods have been deprecated in Java 2, version 1.4, but they're all over the Java map—no particular or important part of Java has been deprecated. What has been deprecated was removed to make it easier to implement the new changes in this version. For a full list of what has been deprecated, go to **http://java.sun.com/j2se/1.4/docs/api/index.html** and click the Deprecated link at the top.

What's new in Java 2, version 5

"OK", says the Novice Programmer, "it's time to get fully updated. What's new in Java 2, version 1.5?" "Plenty", you say. "Let's take a look".

Unlike earlier versions, there are a few major changes that take place in Java 2, version 1.5. Here's an overview of what is new in Java 2, version 1.5.

Caching Entire BufferedImages

In J2SE 5.0, all images created with a **BufferedImage** constructor are now managed images that can be cached in video memory or, in the case of a remote X server, on the X server side. Previously, the Sun implementation managed only compatible images—those created with the **Component createImage(int,**

int) method or with the GraphicsConfiguration createCompatibleImage methods. Managed images generally perform better than unmanaged images.

Methods Related to Hardware Acceleration of Images

The **Image** class has three new methods related to hardware acceleration. The **getCapabilities** method, formerly defined only in **VolatileImage**, allows you to determine whether the image is currently accelerated. Two other methods let you set or get a hint about how important acceleration is for the image: **setAccelerationPriority** and **getAccelerationPriority**.

The GraphicsConfiguration class has two new methods—createCompatibleVolatileImage(int, int, int) and createCompatibleVolatileImage(int, int, ImageCapabilities, int), that allow you to create transparent VolatileImages.

In J2SE 5.0, these methods are not fully operational. While the value set by **setAccelerationPriority** is ignored, the images created with the **createCompatibleVolatileImage** methods are not always hardware accelerated.

On Linux and Solaris systems, only **OPAQUE VolatileImages** are hardware accelerated. On Microsoft Windows systems, images created with **createCompatibleVolatileImage** in J2SE 5.0 are hardware accelerated only if the hardware supports acceleration and one of the following is true:

- The transparency value is **OPAQUE**.
- The transparency value is **TRANSLUCENT**. The translucency acceleration has been specifically enabled at runtime **(sun.java2d.translaccel=true)**.

We expect to fully implement these methods on all platforms in future releases.

Hardware-Accelerated Rendering Using OpenGL

J2SE 5.0 has a new OpenGL-based pipeline for Java 2D. This pipeline provides hardware acceleration for simple rendering operations (text, images, lines, and filled primitives) as well as those that involve complex transforms, paints, composites, and clips. Available on all platforms (Solaris, Linux, and Microsoft Windows), this is currently disabled by default.

In order to silently enable the OpenGL-based pipeline, specify the following system property on the command line:

```
-Dsun.java2d.opengl=true
```

In order to receive verbose console output about whether the OpenGL-based pipeline is initialised successfully for a particular screen, specify "**True**" (note the uppercase T):

```
-Dsun.java2d.opengl=True
```

The minimum requirements for Solaris/Linux are as follows:

- Hardware-accelerated OpenGL/GLX libraries installed and configured properly
- OpenGL version 1.2 or higher
- GLX version 1.3 or higher
- At least one TrueColor visual with an available stencil buffer

Minimum requirements for Microsoft Windows are:

- Hardware-accelerated drivers supporting the WGL_ARB_pbuffer, WGL_ARB_render_texture, and WGL_ARB_pixel_format extensions
- OpenGL version 1.2 or higher
- At least one pixel format with an available stencil buffer

Solaris OpenGL Notes

Accelerated OpenGL libraries for the Solaris SPARC platform are available directly from Sun: http://wwws.sun.com/software/graphics/opengl/index.html

The following Sun framebuffers are known to work with the OpenGL-based Java 2D pipeline:

- Expert3D
- Expert3D Lite
- XVR-500
- XVR-600
- XVR-1000
- XVR-1200

Accelerated OpenGL libraries for the Solaris x86 platform are not available from Sun. However, third parties such as Xi Graphics are known to support OpenGL libraries for Solaris x86.

Notes on Linux OpenGL

Most Linux distributions include the Mesa 3D graphics library, which is a software implementation of the OpenGL specification. Since Mesa does not take advantage of hardware acceleration, it is likely that the OpenGL-based Java 2D pipeline will run much more slowly than the default (X11-based) pipeline. Therefore, to achieve optimal performance with the OpenGL-based pipeline, it is recommended that you install accelerated OpenGL drivers provided by your graphics hardware manufacturer.

Microsoft Windows OpenGL-based Pipeline

To achieve optimal performance with the OpenGL-based pipeline, install accelerated OpenGL drivers provided by your graphics hardware manufacturer.

Support for CUPS Printers— Solaris and Linux

Solaris and Linux systems can now use printers configured as CUPS (Common UNIX Printing System) printers. This expands the printers the Java platform can use with all those supported by CUPS—including most PostScript and raster printers—making it much easier to use low-cost printers with Linux. (CUPS is based on IPP, Internet Printing Protocol.)

Bicubic Interpolation Assistance

The 2D implementation now supports bicubic interpolation and uses it whenever requested. Previously, the **VALUE_INTERPOLATION_BICUBIC** hint defined by the **RenderingHints** class wasn't honoured, and bilinear interpolation was used instead. Now the bicubic rendering hint is honoured, and a new constant **TYPE_BICUBIC** has been added to **AffineTransformOp**.

Generate Fonts from Files and Streams

It is now possible to create **Font** objects from Type 1 fonts and to create **Font** objects directly from files containing either Type 1 or TrueType font data.

To support the new functionality, the **Font** class has a new **createFont** method that creates **Font** objects from files. The pre-existing **createFont** method creates **Font** objects from streams. A new constant, **Font.TYPE1_FONT**, specifies Type 1 fonts to either **createFont method**.

Better Text Rendering Performance and Reliability

A number of internal changes to text rendering code have greatly improved its robustness, performance, and scalability.

Rendering Multilingual Text

2D text rendering using logical fonts now takes advantage of installed host OS fonts for all supported writing systems to render multilingual text. For example, if you run in a Thai locale environment but have Korean fonts installed, both Thai and Korean are rendered.

What has been Deprecated in Java 2, version 1.5?

"Well", says the Novice Programmer, "I've installed Java 2, version 1.5 now, and I'm just about ready to go—but first, what's been deprecated?" "Plenty", you say.

In fact, a number of classes, fields, interfaces, exceptions, constructors and methods have been deprecated in Java 2, version 1.5, but they're all over the Java map—no particular or important part of Java has been deprecated. What has been deprecated was removed to make it easier to implement the new changes in this version. For a full list of what has been deprecated, go to **http://java.sun.com/j2se/1.5.0/docs/api/index.html** and click the Deprecated link at the top.

Before starting the discussion of writing code in this new version of Java 2, that is, version 1.5, we need to know a few very important technologies that are going to have their impact on the functions of Java. Here we will provide a brief introduction of those technologies.

The Novice Programmer says, "When we are upgrading from the version 1.3 to 1.5, some new technologies may have come up". You say yes there are few or rather to say many new technologies evolved, we will go through them in brief here.

What is JSP?

Java Server Pages (JSP) technology allows Web developers and designers to rapidly develop and easily maintain, information-rich, dynamic Web pages that leverage existing business systems. As part of the Java technology family, JSP technology enables rapid development of Web-based applications that are platform-independent. JSP technology separates the user interface from content generation, enabling designers to change the overall page layout without altering the underlying dynamic content.

JSP technology uses XML-like tags that encapsulate the logic that creates the content for the page. The application logic can reside in server-based resources (such as Java Beans component architecture) that the page accesses with these tags. By separating the page logic from its design and display and supporting a reusable component-based design, JSP technology makes it faster and easier than ever to build Web-based applications.

Java Server Pages technology is an extension of the Java Servlet technology. Servlets are platform-independent, server-side modules that fit seamlessly into a Web server framework and can be used to extend the capabilities of a Web server with minimal overheads, maintenance, and support. Unlike other scripting languages, servlets involve no platform-specific consideration or modifications; they are application components that are downloaded, on demand, to the part of the system that needs them. Together, JSP technology and servlets provide an attractive alternative to other types of dynamic Web scripting/programming by offering platform independence, enhanced performance, separation of logic from display, ease of administration, extensibility into the enterprise, and, most importantly, ease of use.

The JSP specification is a product of industry-wide collaboration with industry leaders in the enterprise software and tools markets, led by Sun Microsystems. Sun has made the JSP specification freely available to the developer community, with the goal that every Web server and application server will support the JSP interface. JSP pages share the "Write Once, Run Anywhere" advantage of Java technology. The current JSP version used in various applications is JSP 2.0.

What is J2EE?

J2EE (Java 2 Platform, Enterprise Edition) is a platform-independent, Java-centric environment from Sun for developing, building and online deployment of Web-based enterprise applications. The J2EE platform consists of a set of services—APIs, and protocols—that provide the functionality for developing multi-tiered, Web-based applications.

J2EE is a Java-based, runtime platform created by Sun Microsystems used for developing, deploying, and managing multi-tier server-centric applications on an enterprise-wide scale. It builds on the features of J2SE and adds distributed communication, threading control, scalable architecture, and transaction management. A strong competitor to the Microsoft's .NET Framework, J2EE is a Java platform designed for the mainframe-scale computing typical of large enterprises. Sun Microsystems (together with industry partners such as IBM) designed J2EE to simplify application development in a thin client tiered environment.

Some of the key features and services of J2EE (1.4) are as follows:

- At the client tier, J2EE supports pure HTML (Hypertext Markup Language), 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 Java Beans (EJBs) provide another layer where the platform's logic is stored. An 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 ODBC, is a standard interface for Java databases.
- The Java servlet API enhances consistency for developers without requiring a graphical user interface (GUI).

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

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

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, for example, Tomcat, Jrun, WebSphere, WebLogic, etc.

- 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 collaboration of the best-of-breed developers from around the world.

- Jrun—JRun is an application server from Macromedia that is based on Sun Microsystems' Java 2 Platform, Enterprise Edition. JRun consists of Java Server Pages, Java Servlets, Enterprise Java Beans, the Java Transaction Service (JTS), and the 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 licensed for development use only and is limited to three concurrent connections. The Advanced Edition is designed for deploying JSP and servlet applications in a clustered-server environment. Companies 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 BEA WebLogic Platform™ increases productivity and lowers total cost of ownership for enterprise IT organizations by providing a unified, simplified, 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 application-to-application communication.

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

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 J2EE EAR files, these archives are not considered when the enterprise applications are deployed. These are a part of J2EE applications but do not follow the run time activation process that web application and EJB containers go through during deployment.

The application client archives operate within the content of a J2EE container, and are not deployed into an application server. The application client program runs stand alone, 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 the target environment of the application server. Some of the commonly used deployers are Ant, Junit, Cactus and Maven etc.

- 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. The purpose of the Cactus is to lower the cost of writing tests for server side codes.

Maven is based on the concept of project object model (PMO), which provides a well-defined project structure, well-defined development process and cohesive body of documentation that keeps the developer update regarding the project progress.

What is JSF?

The JavaServer Faces (JSF) is a new framework for building Web applications using Java. JavaServer Faces consist of the following features:

- Easily configurable page navigation
- Standard extensible user interface widgets (like hyperlinks, buttons) for:
  - User input validation
  - Easy error handling
  - Java bean management
  - Event-handling
  - Internationalisation support

What is J2ME?

J2ME (Java 2 Mobile Edition), a Java-based runtime platform created by Sun Microsystems, allows Java applications to run on embedded devices, such as cellular telephones and Personal Digital Assistants (PDAs). J2ME also is a competitor to the Microsoft .NET Compact Framework.

The J2ME provides a robust, flexible environment for applications running on consumer devices, such as mobile phones, PDAs, and TV set-top boxes, as well as a broad range of embedded devices. Like its counterparts for the enterprise (J2EE), desktop (J2SE) and smart card (Java Card) environments, J2ME includes Java virtual machines and a set of standard Java APIs defined through the Java Community Process, by expert groups whose members include leading device manufacturers.

The J2ME delivers the power and benefits of Java technology to that consumer and embedded devices. It includes flexible user interfaces, a robust security model, a broad range of built-in network protocols, and extensive support for networked and offline applications that can be downloaded dynamically. Applications based on J2ME specifications are written once for a wide range of devices, yet exploit each device's native capabilities.

The architecture of J2ME comprises of a variety of configurations, profiles, and optional packages that implementers and developers can choose from, and combine to construct a complete Java runtime environment that closely fits the requirements of a particular range of devices and a target market. Each combination is optimised for the memory, processing power, and I/O capabilities of a related category of devices. The result is a common Java platform that takes full advantage of each type of device to deliver a rich user experience.

The configuration of J2ME comprises of a virtual machine and a minimal set of class libraries. They provide the base functionality for a particular range of devices that share similar characteristics, such as network connectivity and memory footprint. Currently, there are two J2ME configurations: the Connected Limited Device Configuration (CLDC) and the Connected Device Configuration (CDC).

What is Struts?

Struts is a framework that promotes the use of the Model-View-Controller architecture for designing large-scale applications. It is a project of the Apache Software Foundation with its official home page is at **http://struts.apache.org/.**

The Struts framework includes a set of custom tag libraries and their associated Java classes, along with various utility classes. The most powerful aspect of the Struts framework is its support for creating and processing Web-based forms.

The core of the Struts framework is a flexible control layer based on standard technologies like Java Servlets, Java Beans, Resource Bundles, and XML, as well as various Jakarta Commons packages.

Struts provide its own Controller component and integrate with other technologies to provide the Model and the View. For the Model, Struts can interact with standard data access technologies, like JDBC and EJB, as well as most any third-party packages, like Hibernate, iBATIS, or Object Relational Bridge. For the View, Struts works well with JavaServer Pages, including JSTL and JSF, as well as Velocity Templates, XSLT, and other presentation systems.

The Struts framework provides the invisible underpinnings every professional Web application needs to survive. It helps you to create an extensible development environment for your application based on published standards and proven design patterns.

Thus, in a nutshell we can say that Struts is a Web application that uses a deployment descriptor to initialise resources like servlets and taglibs. The deployment descriptor is formatted as a XML document and named "Web.xml". Likewise, Struts uses a configuration file to initialise its own resources. These resources include ActionForms to collect input from users, ActionMappings to direct input to server-side Actions, and ActionForwards to select output pages. The above-mentioned features are covered in version 1.2.6. This will be discussed later in this book.

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, but 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.

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	It contains files to integrate java with other languages.
jre	Runtimes Environment files are included.
lib	Library files, including the Java API class library.
Src	The source code for the java API classes. This folder is only created if you unpack the src.zip file. It would make you to learn more about how the API classes work.

Table 1.2 Additional files in JDK root folder.

File	Description
README>html	The Java readme file in HTML format.
README.txt	This is the text format of the readme file.
LICENSE	The Java license that you agreed to when you downloaded the JDK.
LICENSE.rtf	This is also a license file stated above but in RTF format which can be understood by word processing programs.
COPYRIGHT	It is the copyright notice about all the copyright and export laws applied by Sun

Writing Code: Creating Code 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, for example, 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're 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 a few pages.

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

Writing Code: Knowing Java's Reserved Words

The Novice Programmer appears and says, "Java is acting all funny—I want to name a variable '**public**', but it's giving me all kinds of problems". "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're writing Java code, you should know that Java reserves certain words for itself as part of the Java language. There aren't too many of them, though. Here they are (I'll cover these keywords throughout the book):

- **abstract**—Specifies that a class or method will be implemented later, in a subclass
- **boolean**—A data type that can hold True and False values only
- **break**—A control statement for breaking out of loops

- **byte**—A data type that can hold 8-bit data values
- **byvalue**—Reserved for future use
- **case**—Used in **switch** statements to mark blocks of text
- **cast**—Reserved for future use
- **catch**—Catches exceptions generated by **try** statements
- **char**—A data type that can hold unsigned 16-bit Unicode characters
- **class**—Declares a new class
- **const**—Reserved for future use
- **continue**—Sends control back outside a loop
- **default**—Specifies the default block of code in a **switch** statement
- **do**—Starts a **do-while** loop
- **double**—A data type that can hold 64-bit floating-point numbers
- **else**—Indicates alternative branches in an **if** statement
- **extends**—Indicates that a class is derived from another class or interface
- **final**—Indicates that a variable holds a constant value or that a method will not be overridden
- **finally**—Indicates a block of code in a **try-catch** structure that will always be executed
- **float**—A data type that holds a 32-bit floating-point number
- **for**—Used to start a **for** loop
- **future**—Reserved for future use
- **generic**—Reserved for future use
- **goto**—Reserved for future use
- **if**—Tests a true/false expression and branches accordingly
- **implements**—Specifies that a class implements an interface
- **import**—References other classes
- **inner**—Reserved for future use
- **instanceof**—Indicates whether an object is an instance of a specific class or implements a specific interface
- **int**—A data type that can hold a 32-bit signed integer
- **interface**—Declares an interface
- **long**—A data type that holds a 64-bit integer
- **native**—Specifies that a method is implemented with native (platform-specific) code
- **new**—Creates new objects
- **null**—Indicates that a reference does not refer to anything
- **operator**—Reserved for future use
- **outer**—Reserved for future use
- **package**—Declares a Java package
- **private**—An access specifier indicating that a method or variable may be accessed only in the class it's declared in
- **protected**—An access specifier indicating that a method or variable may only be accessed in the class it's declared in (or a subclass of the class it's declared in or other classes in the same package)
- **public**—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**—Reserved for future use

- **return**—Sends control and possibly a return value back from a called method
- **short**—A data type that can hold a 16-bit integer
- **static**—Indicates that a variable or method is a class method (rather than being limited to one particular object)
- **super**—Refers to a class's base class (used in a method or class constructor)
- **switch**—A statement that executes code based on a test value
- **synchronized**—Specifies critical sections or methods in multithreaded code
- **this**—Refers to the current object in a method or constructor
- **throw**—Creates an exception
- **throws**—Indicates what exceptions may be thrown by a method
- **transient**—Specifies that a variable is not part of an object's persistent state
- **try**—Starts a block of code that will be tested for exceptions
- **var**—Reserved for future use
- **void**—Specifies that a method does not have a return value
- **volatile**—Indicates that a variable may change asynchronously
- **while**—Starts a **while** loop

Writing Code: 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 I'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're new to Java, this might look strange to you. The idea here is that this application will print out 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.

public class app

Here's the first line in app.java:

```
public class app
{
    .
```

```
    .
    .
}
```

This line indicates that we're creating a new Java class named **app**. After we translate this class into bytecodes, the Java Virtual Machine 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 it in Chapters 4 and 5. The **public** access specifier indicates that this class is available anywhere in a program that makes use of it.

Note also 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 the 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, which normally doesn't pay any attention to capitalization).

The actual implementation of the class we're defining here will go between the curly braces:

```
public class app
{
    .
    .
    .
}
```

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

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're creating a *method* in the **app** class. A method in object-oriented programming 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, the Java Virtual Machine executes the code in the method.

You'll be introduced to methods formally in Chapter 4, but here the idea is that we're creating a method named **main**, which is the method that the Java Virtual Machine 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, as you'll see in Chapter 4, that **main** is a class method, not an object method. 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're indicating that **main** is 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**, then "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.

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

The main method has one line of code in it:

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

This is the actual line of code that does all the work. In this case, we're 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.

Note also 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're 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 to 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 Java Virtual Machine can use, you use the Java compiler, which is called **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]
```

Here are the arguments to javac:

- *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 this 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:\Edward, then to compile app.java, first make sure C:\Edward is the current directory (you can use the change directory command, **cd**, like this: **cd C:\Edward**). Then run javac, like this: **C:\Edward>javac app.java**. Alternatively, you can specify the path of app.java, like this: **C:\>javacC:\Edward\app.java**.

Also, note that in this book, I'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/steve/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 and (assuming there are no errors) compiles it, translating it and creating 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've forgotten the **println** method and tried to use one called **printline**:

```
C:\>javac app.java
app.java:5: Method printline(java.lang.String) not found in class
java.io.Print
Stream.
    System.out.printline("Hello from Java!");
                         ^
1 error
```

When app.java is successfully compiled to bytecodes, the new file, app.class, contains all that the Java Virtual Machine will need to create objects from the **app** class. So, now we've created app.class. Now, how do you actually run it in the JVM? See the next solution.

Compiling Code: Using Command-Line Options

"Hmm", says the Novice Programmer, "I've 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". "Swell", 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 subdirectory of the current directory:

```
javac -d temp app.java
```

Here's 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**—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***—Sets the destination directory for class files. For readers who know what Java packages are, if a class is part of a package, javac puts the class file in a subdirectory 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**—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**—Sets the source file encoding name. If **-encoding** is not specified, the platform default converter is used.
- **-g**—Generates all debugging information, including local variables. By default, only line number and source file information is generated.
- **-g:none**—Makes the compiler not generate any debugging information.
- **-g:*[keyword list]***—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**—Disables all warning messages.
- **-O**—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 **-g** and **-O**, but you might get odd results, such as missing variables and relocated or missing code. **-O** no longer automatically turns on **-depend** or turns off **-g**.
- -sourcepath *sourcepath*—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**—Creates "verbose" output. This includes information about each class loaded and each source file compiled.
- **-X**—Displays information about non-standard options and quits.
- **-Xdepend**—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**—Sends compiler messages to **System.out**. By default, the compiler messages go to **System.err**, which you'll learn more about later.
- **-Xverbosepath**—Describes how paths and standard extensions were searched to find source and class files.
- **-J*option***—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*—Generates class files that will work on JVMs with the specified version.
- **-bootclasspath** *bootclasspath*—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*—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

"Jeez", says the Novice Programmer, "how can I keep all the deprecated methods in Java straight? Now that I'm upgrading to Java 2, version 1.5, 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 if you're using a method that's been deprecated, which it didn't used to do 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 I use to make sure I avoid deprecated methods. Suppose you have a 200-line program, and when you try to compile it, javac gives you this result:

```
C:\>javac app.java
Note: app.java uses or overrides a deprecated API. Recompile with "-
deprecation" for details.
1 warning
```

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

```
C:\>javac app.java -deprecation
app.java:109: Note: The method java.awt.Dimension size() in class
```

```
java.awt.Component has been deprecated.
   x = (size().width - fontmetrics.stringWidth(text)) / 2;
                ^
Note: app.java uses or overrides a deprecated API. Please consult the
documentation for a better alternative.
1 warning
```

As you can see, the problem is that app.java uses the **size** method in line 109, and that method has been deprecated; replacing it with the new version, **getSize**, solves the problem.

Running Code

The Big Boss is getting impatient. You've written a new application and compiled it without errors 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 the JVM. Here's how you can use the java tool:

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

Here are the parameters in the preceding lines:

- ***options***—Command-line options, which I'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 28.
- ***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):

```
java app
```

The result appears immediately:

```
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, then 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 C:\Nancy\app****.*

You can see how this works in a DOS window under Windows in Figure 1.4.

Figure 1.4 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 shown in Figure 1.5.

There's one thing to note about Figure 1.5—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:

```
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 ...]
```

Here are the parameters used by javaw:

- *options*—Command-line options, which I'll cover later in the chapter.
- *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.

Figure 1.5 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're 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.

In fact, there's another launcher in Java 2—the oldjava launcher, which Sun included for backward compatibility. The oldjava launcher does not support the Java Extensions Framework. It gives you backward compatibility when you have an application that uses a Java 1.1-style security manager, which is incompatible with the 1.2 or later class-loading techniques. We're sticking to Java 2 in this book, so there won't be much use for oldjava, but if you're migrating to Java 2 and need to use the old class-loading mechanism, it's good to know it's there. There's also an oldjavaw tool.

That's how you run an application—you use a Java tool such as java, javaw, or oldjava. When you launch a Java 2 application, the Just In Time compiler in the launcher compiles the bytecodes in sections and runs the application.

***TIP:** If you don't want to use the JIT compiler for some reason, there are two ways to disable it. You can set the environment variable **JAVA_COMPILER** to **NONE**, using, for example, the **SET** command in Windows 95/98/Millennium/XP or the System Control Panel in Windows 2000. You can also use the **-D** command-line option to set the keyword **java.compiler** to **NONE**, like this: **java -Djava.compiler=NONE app**.*

While we're 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 Novice Programmer, "I have another problem. I've 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 "Basic Skills: 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, javaw, and oldjava tools, like this (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 ...]
oldjava [options] class [argument ...]
oldjavaw [options] class [argument ...]
```

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

- **-classpath classpath** *or* **-cp classpath**—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. Used with oldjava or oldjavaw, **-classpath** or **-cp** specifies the class path for both user classes and bootstrap 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 "Basic Skills: Finding Java Classes with **CLASSPATH**", later in this chapter, for more information.
- **-Dproperty=value**—Sets a system property value.
- **-jar**—Executes a program encapsulated in a JAR file. The first argument is the name of a JAR file instead of a startup class name. When you use this option, the JAR file is the source of all user classes, and other user class path settings are ignored. The oldjava and oldjavaw tools do not support the **-jar** option.
- **-verbose** *or* **-verbose:class**—Displays information about each class loaded.
- **-verbose:gc**—Reports on each garbage-collection event. Garbage collection involves automatic memory management in Java.
- **-verbose:jni**—Reports information about the use of native (that is, platform specific) methods and other Java Native Interface activity.
- **-version**—Displays version information and exits.
- **-?** *or* **-help**—Displays usage information and exits.
- **-X**—Displays information about non-standard options and exits.
- **-Xbootclasspath:bootclasspath**—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**—Starts with the debugger enabled.
- **-Xnoclassgc**—Disables class garbage collection.
- **-Xmsn**—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**—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>,...]**—Enables CPU, heap, or monitor profiling. This option is usually followed by a list of comma separated pairs of the form **<suboption>=<value>**.
- **-Xrs**—Reduces the use of operating system signals.
- **-Xcheck:jni**—Performs additional checks for Java Native Interface functions.
- **-Xfuture**—Performs strict class-file format checks.

Basic Skills: 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, 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've 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 comments, two of which are taken from C++. You can surround a comment of any length with the characters "**/***" and "***/**", like this:

```
/* 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, like this:

```
/* 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're expected to use a standard comment header, created with the /* and */ forms of comment, for all new code. It might look something like this:

```
/*********************************************************
* 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, like this:

```
/* 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. We'll take a look at this in Chapter 22. Here's an example using /** and */:

```
/** 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.

Basic Skills: Import Java Packages and Classes

"Hmm", says the Novice Programmer, "I've got a problem. The Design Team Coordinator told me to use the **Date** class to print out 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 Java 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're going to use **import** statements to import classes into a program, the **import** statements should be at the top of the code. Now we're free to use the **Date** class, like this (note that we're creating an object from the **Date** class 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, like this:

```
C:\>java app
```

```
Today = Mon Aug 02 12:15:13 EDT 2000
```

As you can see 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, as in this case, where we're 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 this way (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();
    }
}
```

This works just as it should. Congratulations, you've just imported a class into a program. This technique is fine if **printer.class** is in the same directory in which you're compiling this application, because the Java compiler will search the current directory by default. 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**.

Basic Skills: Finding Java Classes with CLASSPATH

"That darn Johnson", the Novice Programmer 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; see the solution "What's New in Java 2, Version 1.5?" in this chapter), and classes in the current directory (that is, where you're compiling your program). Classes can be stored in .class files, in .jar (Java Archive) files, and .zip files. Java can search all these types of files.

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!");
    }
}
```

Now say, as in the previous solution that you want to use the **print** method in another class—as in this case, where we're creating a new object of the **printer** class 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're 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 one in Windows 95/98/Millennium (note that it's important here not to have any spaces around the equals sign):

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

You can use this line at the DOS prompt or in autoexec.bat so the **CLASSPATH** environment variable will be set automatically each time a DOS window is opened. In Windows 2000, you can enter this line at the DOS prompt or follow these steps so the **CLASSPATH** environment variable will be set automatically:

1. Open the Start menu and select Settings | Control Panel. Double-click the System icon.
2. In the System Properties dialog box, click the Advanced tab, followed by the Environment Variables button.
3. Click the **CLASSPATH** variable. Use the New button if necessary to create a new **CLASSPATH** variable.
4. Edit the **CLASSPATH** setting the way you want it and click OK.

You can also determine the current setting of **CLASSPATH** using the **SET** command by itself:

```
C:\>SET
TMP=C:\WINDOWS\TEMP
PROMPT=$p$g
winbootdir=C:\WINDOWS
COMSPEC=C:\WINDOWS\COMMAND.COM
PATH=C:\WINDOWS;C:\JDK1.5\BIN
windir=C:\WINDOWS
CLASSPATH=C:\CLASSES;C:\NEWCLASSES
```

Now the Java compiler (and other Java tools, such as the java tool) will know enough to search c:\classes and c:\newclasses automatically. That 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**.

Creating Applets

The Big Boss is getting impatient. "What's all this about applications that print out 'Hello from Java!' in the console window? What we want to use Java for is to create applets that you can look at in Web browsers". "OK", you say, "just give me a minute".

In the following chapters, we'll take a look at the Java syntax, which might make it a hard path for you to follow if you're primarily interested in writing applets. What's more, it seems intolerable that we shouldn't start out a book on a language as visual as Java without at least one applet. Therefore, in this solution, I'll cover the process of creating a Java applet in overview. Knowing how to create a basic applet will help if you want to test the syntax of the next few chapters visually. Applets will be formally introduced in Chapter 7, so consider this as a sneak preview.

Standard applets are built on the **Applet** class, which is in the **java.applet package.** Therefore, we'll start by importing that class in a new Java source code file, which we'll call applet.java:

```
import java.applet.Applet;
    .
    .
    .
```

The **java.applet.Applet** class is the class that forms the base for standard applets, and you can derive your own applet classes from this class using the extends keyword:

```
import java.applet.Applet;
    public class applet1 extends Applet    //applet is reserved
    {
        .
        .
        .
    }
```

So far, so good; now it's time to add code to this new applet. Applets don't have a **main** method like applications do—in fact, that's the primary code difference between applets and applications. So, how can you display text directly in an applet?

The actual drawing of an applet is accomplished in its **paint** method, which the Java Virtual Machine calls when it's time to display the applet. The **java.applet.Applet** class has its own **paint** method, but we can *override* that method by defining our own **paint** method, like this (see Chapters 4 and 5 for details on overriding):

```
import java.applet.Applet;
import java.awt.*;
```

```
public class applet1 extends Applet
{
    public void paint(Graphics g)
    {
        .
        .
        .
    }
}
```

This method, **paint**, is actually a part of the Java Abstract Windowing Toolkit (AWT) that you'll see a great deal of in this book, so we've imported the AWT classes with the statement **import java.awt.*** here. You'll see how the following details work later in this book, but for now, here's basically how they work: The **paint** method is passed a Java object of the **Graphics** class (this object is named **g** in the code). You can use this object's **drawString** method to actually draw the text. In this case, we'll draw the text "Hello from Java!" at location (60, 100) in the applet; coordinates are measured in pixels from the upper-left corner of the applet, so this position is 60 pixels from the left border of the applet and 100 pixels from the top. Here's what the code looks like:

```
import java.applet.Applet;
import java.awt.*;
public class applet1 extends Applet
{
    public void paint(Graphics g)
    {
            g.drawString("Hello from Java!", 60, 100);
    }
}
```

That's all it takes; now you can compile applet1.java to applet1.class. There's one more step to take—creating a Web page to display the applet in. We'll take a look at that next.

Running Applets

"OK", the Big Boss says, "you've created an applet. Why don't I see it in a Web page?" "Coming right up", you say. "I think..."

To display an applet, you can use a Web page with an HTML <APPLET> tag in it. In fact, you can use a shortcut in which you actually store the needed HTML in the applet's source code file, as you'll see in Chapter 7 (you'll also learn all about the <APPLET> tag in that chapter). For now, here's a Web page, applet.html, which will display the applet developed in the previous solution:

```
<HTML>
<HEAD>
<TITLE>APPLET</TITLE>
</HEAD>
<BODY>
<HR>
<CENTER>
<APPLET
    CODE=applet1.class
```

```
    WIDTH=200
    HEIGHT=200 >
</APPLET>
</CENTER>
<HR>
</BODY>
</HTML>
```

You can open this applet Web page in a Web browser, as shown in Figure 1.6, where the applet is opened in Microsoft Internet Explorer. You can also use the Sun appletviewer, which comes with Java, to open applet.html, like this:

```
C:\>appletviewer applet1.html
```

Figure 1.7 shows the applet in the Sun applet viewer.

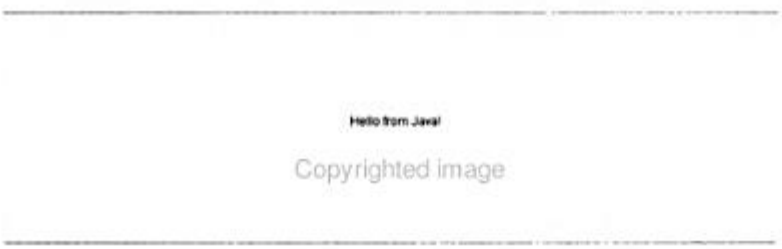

Figure 1.6 An applet at work in Internet Explorer.

Figure 1.7 An applet at work in the Sun applet viewer.

Here's an important point: Your applets might not run in current browsers because those browsers don't implement the latest version of Java yet. To fix this problem, you should use the Java Plug-in; see Chapter 7.

Creating Windowed Applications

The Big Boss is impressed with your new applet and asks, "Can you also make an application display window?" "Sure", you say. "Coming right up".

You'll learn all about creating windowed applications in depth in Chapter 7, but it's worth taking a sneak preview here. Creating a windowed application is much like creating an applet, except that you have to have a **main** method, and you're responsible for creating the window yourself. To create the window for the application, we'll derive a new class from the AWT **Frame** class and add the same code to the **paint** method that was used in the applet in the previous solution:

```
import java.awt.*;
class AppFrame extends Frame
{
    public void paint(Graphics g)
    {
            g.drawString("Hello from Java!", 60, 100);
    }
}
```

Now we'll create the application class itself, which we'll name app. This is the class that will have a **main** method, and in that method, we'll use the new operator to create a new object of the **AppFrame** class, give it a size in pixels, and show it on the screen, all of which looks like this:

```
import java.awt.*;
import java.awt.event.*;
class AppFrame extends Frame
{
    public void paint(Graphics g)
    {
            g.drawString("Hello from Java!", 60, 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.show();
     }
}
```

TIP: Including the line of code here having to do with the addWindowListener method, earns that when the application window is closed, the application itself will end. You'll learn more about how this very compact and powerful line, which uses inner classes and adapter classes, does its work in Chapter 7 (see "Exiting an Application When Its Window Is Closed" in that Chapter).

Now that the new windowed application is ready to go, how do you actually run it? Take a look at the next solution.

Running Windowed Applications

As with console applications, to run a windowed application, you can use the java or javaw tool, like this:

```
java app
javaw app
```

The java tool launches the application and makes the console window wait until the application is dismissed, whereas the javaw tool launches the application and doesn't wait until the application is dismissed. The running application appears in Figure 1.8.

That's all there is to it—now you're running windowed Java applications.

Figure 1.8 A windowed application.

Designing Java Programs

You've been made the head of program design—and your new office is a stunner. But as you sit there, gazing out of your corner window and stroking your new teak desk, you wonder if you can handle the new position.

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 optimisation 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 of 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're 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 when 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 I'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 larger tasks and larger 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. Extensibility is usually a concern only with larger applications, and it often involves an entire interface especially designed for extension modules. In fact, Java itself is designed to be extended using the Java Extension Framework.

Availability

Availability is the measure of how much of the time your application can be used in comparison to the time users want to use it. 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, 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 I won't go into more detail here. But it's worth noting that programmers shouldn't short-change the crucial design steps because—in serious projects—that can lead to more problems in the long run than time saved in the short run.

Distributing Your Java Program

"Well", the Novice Programmer says, "I've finished my Java program, and I'm ready to sell it". "Oh yes?" you ask. "Better check the licensing agreement first".

In order for users to run your programs, they'll need to have a Java runtime environment on their systems. The Java 2 SDK contains a runtime environment, so users could use that if they happen to have it installed. However, note that most users won't have the whole Java 2 SDK installed, so a better choice for your users will be the Java 2 Runtime Environment (JRE). Here's why distributing the JRE instead of the SDK is a good idea:

- The Java 2 Runtime Environment is redistributable, and the Java 2 SDK is not, which is to say that the JRE license lets you to package it with your software. By distributing the JRE with your application, you can make sure your users have the correct version of the runtime environment for your software.
- The JRE is also smaller than the SDK. The JRE contains all that users need to run your software, but it doesn't include the development tools and applications that are part of the SDK. Because the JRE is small, it's easier for you to package with your software as well as for users to download for themselves from the Java Software Web site.
- In Windows, the JRE installer automatically installs java and javaw in the operating system's path, which means you don't have to worry about finding the launchers to start your application (this, in turn, means that you don't have to give instructions to users for setting up paths on their systems).

You can find more information about the version 1.5 runtime environment at **http://java.sun.com/j2se/1.5/jre/index.html**.

Variables, Arrays, and Strings

If you need an immediate solution to:	*See page:*
What Data Types Are Available?	55
Creating Integer Literals	56
Creating Floating-Point Literals	57
Creating Boolean Literals	58
Creating Character Literals	58
Creating String Literals	60
Declaring Integer Variables	60
Declaring Floating-Point Variables	61
Declaring Character Variables	62
Declaring Boolean Variables	63
Initializing Variables	64
Dynamic Initialisation	65
Converting between Data Types	65
Automatic Conversions	66
Casting to New Data Types	66
Declaring One-Dimensional Arrays	68
Creating One-Dimensional Arrays	69
Initializing One-Dimensional Arrays	69
Declaring Multi-dimensional Arrays	70
Creating Multi-dimensional Arrays	71
Initialising Multi-dimensional Arrays	72
Creating Irregular Multi-dimensional Arrays	73
Getting an Array's Length	74

If you need an immediate solution to:	See page:
General form of Static Import	75
Importing Static Members	76
The **String** Class	76
Creating Strings	80
Getting String Length	82
Concatenating Strings	83
Getting Characters and Substrings	83
Searching for and Replacing Strings	85
Changing Case in Strings	86
Formatting Numbers in Strings	86
The StringBuffer Class	87
Creating String Buffers	89
Getting and Setting String Buffer Lengths and Capacities	90
Setting Characters in String Buffers	91
Appending and Inserting Using String Buffers	91
Deleting Text in String Buffers	92
Replacing Text in String Buffers	92
Wrapper class	93
Autoboxing and Unboxing of Primitive Types	94
Varargs Fundamentals	98
Overloading Varargs Methods	100
Ambiguity in Varargs	102

In Depth

This chapter begins our discussion of *Java syntax*, and you'll see a great deal of syntax usage in this chapter. I'm 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 Java, take a look at the material in this chapter, because there's a lot coming up.

Variables

Variables come in different types and serve as placeholders in memory for data. The different types of variables are used to store data in different format and they are also chosen taking into account how much memory is to be set aside to hold that data. For example, an integer variable type, the **int** type, is made up of 4 bytes (or 32 bits), and you can use it to store integer values. This gives the data in the **int** type a range of possible values from –2,147,483,648 to 2,147,483,647. There are quite a few different variable types built into Java, such as integers, floating-point numbers, and individual characters, and you'll see them all 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 an integer 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 of 365 in **days**, 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 literals Java allows throughout this chapter. To verify that **days** now holds 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've created a variable, stored data in it, and fetched that data back to print it on the screen. That's how it works.

There's also a convenient shortcut that lets you initialise a variable when you declare it. Here, **days** is declared and initialised 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*—These types are **byte**, **short**, **int**, and **long**, which hold signed, whole-value numbers.
- *Floating-point numbers*—These types are **float** and **double**, which hold signed floating-point numbers.
- *Characters*—This is the **char** type, which holds representations of characters such as letters and numbers.
- *Boolean*—This type is designed to hold 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 into 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 into 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 fine for storing single data items, but data is often more complex than that. Say, for example, that you want to start a new bank, the Java Programming Bank, and need to keep track of the amount of money in every account, as indexed by account number. A method of working with compound data is best here, and that's what arrays provide.

Using an array, you can group simple data types into a more *compound data structure* and refer to that new data structure by name. More importantly, you can refer to the individual data items stored in the array by numeric index. That's important, because computers excel at performing millions of operations very quickly, so if your data may be referenced with a numeric index, you can work through a whole set of data very quickly simply by incrementing the array index and thus accessing all the items in the array.

Here's an example. In this case, I'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 **accounts[]** indicate that it's an array, and you place the index number of the item in the array you want to access in the braces. Here's how I create the **accounts[]** array, making each entry in it of the floating-point type **double** for extra precision. First, I declare the array; then, I 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 I've created an array with 100 items, I can refer to those items numerically, like this (note that I'm 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[3] = 43.95;
        System.out.println("Account 3 has $" + accounts[3]);
    }
}
```

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 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 whose 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[3] = 43.95;
        System.out.println("Account 3 has $" + accounts[3]);
    }
}
```

You can also initialise 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};
        accounts[3] = 43.95;
        System.out.println("Account 3 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, I'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][3] = 43.95;
            accounts[1][3] = 2385489382.06;
            System.out.println("Savings account 3 has $" + accounts[0][3]);
            System.out.println("Checking account 3 has $" + accounts[1][3]);
    }
}
```

Now that **accounts[][]** is a two-dimensional array, each item in it is referred to using two index values; for example, the savings balance for account 3 is now **accounts[0][3]**, and the checking balance is **accounts[1][3]**. Shown below 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 I've 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 $2.38548938206 \times 10^9$—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 2 now supports much more complex data structures than arrays. These data structures are built into the language. Java 2 now supports hashes and maps as well as other types of data structures, 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 I've 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][3] = 43.95;
            accounts[1][3] = 2385489382.06;
            System.out.println("Savings account 3 has $" + accounts[0][3]);
            System.out.println("Checking account 3 has $" + accounts[1][3]);
    }
}
```

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 I 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 I can treat this string as I 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, 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's 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, which 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're 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. The simple data types can be broken up by category, as I did at the beginning of this chapter, like this:

- *Integers*—These are **byte**, **short**, **int**, and **long**. They hold signed, whole-value numbers.
- *Floating-point numbers*—These are **float** and **double**. They hold signed, floating-point numbers.
- *Characters*—This is the **char** type, which holds representations of characters such as letters and numbers.
- *Boolean*—This type is designed to hold only two types of values: true and false.

Table 2.1 Variable types.

Variable Type	Bytes of Storage	Range
Boolean	2	True, False
Byte	1	-128 to 127
Char	2	N/A
double	8	-1.79769313486232E308 to -45841247E-324 for negative values and 4.94065645841247E-324 to 1.79769313486232E308 for positive values
Float	4	-3.402823E38 to -1.401298E-45 for negative values and 1.401298E-45 to 3.402823E38 for positive values
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, starting in Java 2 version 1.5, you can now use the* ***StrictMath*** *classes (you can find more about these classes at* ***http://java.sun.com/j2se/1.5/docs/api/index.html****) to ensure that the same mathematical precision is used no matter what platform your program is running on.*

Related solutions	Found on page
Incrementing and Decrementing: ++ and —	112
Multiplication and Division: * and /	114
Addition and Subtraction: + and –	115
Shift Operators: >>, >>>, and <<	116

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. I've already used integer literals, which are the most common types programmers use, in this chapter. Here's the earlier example:

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

Here, I'm 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's 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 them 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?" "By default", you say, "floating-point numbers you use as literals are of type **double**, 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 I'm 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 -deprecation
app.java:7: Incompatible type for =. Explicit cast needed to convert double
to float.
value = 1.5;
      ^
```

```
1 error
```

I 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 part of Java's strong data typing). This means the *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 **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 alphabet character = " + char3);
    }
}
```

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 alphabet character = " + char3);
    }
}
```

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 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, I'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").

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 we'll go through it".

Java uses four types of integers, 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 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 I 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 I explicitly make each literal value either a **float** or a **double** literal so 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 I've 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 I 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 I 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. I'm going to jump the gun here and give you a sneak peak into the next chapter. Here, I'm using these two Boolean variables with the Java **if** statement. I test the value in **boolean1** with the **if** statement, making 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

"OK", the Novice Programmer says, "I've 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, I've 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, I can combine these two steps into one by initialising a variable when I 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 initialise 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
```

Dynamic Initialisation

Up to this point, I've 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 I'm 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 initialised will be determined at runtime, which is why this process is called *dynamic initialisation*.

As in C++, in Java you can also intersperse your variable declarations throughout your code, as I'm 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
```

Converting 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're often faced with the situation of assigning a variable of one type to a variable of another. You have two ways you can do this: relying on automatic type conversion and making an explicit type cast. We'll take a look at both here.

Automatic Conversions

When you're 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
```

These types of conversions, where you convert to a data type with a larger range, are called *widening conversions*. 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

If you're assigning a data value that's of a type that has a larger range than the variable you're assigning it to, you're performing what's called a *narrowing conversion*. The Java compiler will not perform narrowing conversions automatically because there's the possibility that precision will be lost. If you want to perform a narrowing conversion, you must use an explicit cast, which looks like this:

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

For example, in this code, I'm converting an **integer** type to 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);
    }
}
```

Without the explicit type cast, the compiler would object, but with the type cast, there's no problem, because Java decides that you know about the possibility of losing some data when you cram a possibly larger value into a smaller type. In other words, you're taking responsibility for the results. 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 only involves 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:8: possible loss of precision
found : int
required : 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 us". "OK", you say, "can I see the accounts?" "We 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 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 I'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, I 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[3] = 1335.67;
            System.out.println("Account 3 is overdue by $" + accounts[3]);
    }
}
```

Here, I've created an array of exactly 100 **double** values, all of which Java initialises 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[3] = 1335.67;
            System.out.println("Account 3 is overdue by $" + accounts[3]);
    }
}
```

Initializing One-Dimensional Arrays

The Novice Programmer is back with a question. "I know I can initialise simple variables when I declare them", the Novice Programmer says, "but what about initialising 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 initialises 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 3 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've 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][3] = 43.95;
            accounts[1][3] = 2385489382.06;
            System.out.println("Savings account 3 has $" + accounts[0][3]);
            System.out.println("Checking account 3 has $" + accounts[1][3]);
    }
}
```

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][3] = 43.95;
            accounts[1][3] = 2385489382.06;
            System.out.println("Savings account 3 has $" + accounts[0][3]);
            System.out.println("Checking account 3 has $" + accounts[1][3]);
    }
}
```

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** method (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've 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. But first let us look at initialising multi-dimensional arrays.

Initialising Multi-dimensional Arrays

You can initialise multi-dimensional arrays with data when declaring them in much the same way you initialise one-dimensional arrays—just include a set of curly braces for each dimension and place the values with which you want to initialise the array in those curly braces. For example, here's how to initialise 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][3]);
        System.out.println("Checking account 3 has $" +
        accounts[1][3]);
    }
}
```

Here's what running this code yields:

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

Creating Irregular Multi-dimensional Arrays

"OK", says the Novice Programm0er 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 this 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 I'm 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 one dimensional array defined in this program, fetches the element according to the value of index taken individually. Note here that the values of the array varialble which 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
Account [3][3] has $1335.67
```

Now, lets fetch the value which 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 above program gets compiled but what running code yields, is the exceptional handling error as follows:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 500
    at app45.main(app45.java:12)
```

Getting an Array's Length

It's often useful to know the length of an array, especially if you're 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 += 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
```

General form of Static Import

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

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

In order to get this, you sometimes put static members into an interface and inherit the same from that interface. This should not be encouraged because a class of the static members of another class you are using is just an implementation detail. Actually, whenever a class implements an interface, it becomes part of the class's public API and implementation details should not interfere into public APIs.

Unqualified access to static members *without* inheriting from the type containing the static members is allowed with the static import construct. 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 above and the normal import declaration is comparable. The static import declaration imports static members from classes allowing you to use the same without class qualification unlike for the normal import declaration, which imports classes from packages allowing their use without package qualification.

In short, 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 one or two 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

You can import the names of static members of a class from named packages into your programs. This permits you to reference such static members by their simple unqualified names. For example, in the **area** class, you could have used the constant **PI** that is defined in the **Math** class by using its fully qualified name, **Math.PI**.

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

This shows the need for the static member of the area 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 of the code, and it would be better without it. We can remove the need for prefixing **PI** with the **Math** class name by importing the **PI** member name from the **Math** class:

```
import static java.lang.Math.PI;
class area
{
    .
    .
    .
    double area_circle()
    {
        return PI*radius*radius;
    }
}
```

It is clear what **PI** means here and the code is not cluttered up with the class name prefix.

You can also import all the static members of a class using * notation. The import applies only to classes that are 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've 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, "but, in fact, 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 I 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 to a character array, convert numbers into strings, search strings, create substrings, change the case of the string, get a string's length, compare strings, and much more. The String class is 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 preview of the **String** class's 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's 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.

I'll put the material you see in these tables to use in the next few topics, in which we'll create and use **String** objects.

Table 2.2 String class field summary.

Field	Means
static Comparator CASE_INSENSITIVE_ORDER	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	Means
String()	Initialises a new **String** object so that it holds an empty character sequence.
String(byte[] bytes)	Constructs a new **String** object by converting the array of bytes using the platform's default character encoding.
String(byte[] ascii, int hibyte)	Deprecated. This method does not properly convert bytes into characters.
String(byte[] bytes, int offset, int length)	Constructs a new **String** object by converting the subarray of bytes using the default character encoding.
String(byte[] ascii, int hibyte, int offset, int count)	Deprecated. This method does not properly convert bytes into characters.
String(byte[] bytes, int offset, int length, String enc)	Constructs a new **String** object by converting the subarray of bytes using the specified character encoding.
String(byte[] bytes, String enc)	Constructs a new **String** object by converting the array of bytes using the specified character encoding.

Table 2.3 String class constructor summary.

Constructor	Means
String(char[] value)	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)	Allocates a new **String** object that contains characters from a subarray of the character array argument.
String(String value)	Initialises a new **String** object so that it represents the same sequence of characters as the argument string.
String(StringBuffer buffer)	Allocates a new **String** object that contains the sequence of characters contained in the string buffer argument.

Table 2.4 String class methods.

Method	Means
char charAt(int index)	Yields the character at the given index.
int compareTo(Object o)	Compares this **String** object to another object
int compareTo(String anotherString)	Compares two strings lexicographically.
int compareToIgnoreCase(String str)	Compares two strings lexicographically, ignoring case.
String concat(String str)	Concatenates the given string to the end of this string.
Static String copyValueOf(char[] data)	Yields a **String** object that's equivalent to the given character array.
static String copyValueOf(char[] data, int offset, int count)	Yields a **String** object that's equivalent to the given character array, using offsets.
boolean endsWith(String suffix)	True if this string ends with the given suffix.
boolean equals(Object anObject)	Compares this string to an object.
boolean equalsIgnoreCase(String anotherString)	Compares this **String** object to another String object, ignoring case.
byte[] getBytes()	Converts this **String** object into bytes according to the default character encoding, storing the result in a new byte array.
void getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin)	Deprecated. This method does not properly convert characters into bytes.
byte[] getBytes(String enc)	Converts this **String** object into bytes according to the given character encoding, storing the result in a new byte array.
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	Copies characters from this string into the destination array.
int hashCode()	Yields a hashcode for this string.
int indexOf(int ch)	Yields the index within this string of the first occurrence of the given character.
int indexOf(int ch, int fromIndex)	Yields the index within this string of the first occurrence of the given character, starting at the given index.

Table 2.4 String class methods.

Method	Means
int indexOf(String str)	Yields the index within this string of the first occurrence of the given substring.
int indexOf(String str, int fromIndex)	Yields the index within this string of the first occurrence of the given substring, starting at the given index.
String intern()	Yields a representation for the **String** object.
int lastIndexOf(int ch)	Yields the index within this string of the last occurrence of the given character.
int lastIndexOf(int ch, int fromIndex)	Yields the index within this string of the last occurrence of the given character, searching backward from the given index.
int lastIndexOf(String str)	Yields the index within this string of the rightmost occurrence of the given substring.
int lastIndexOf(String str, int fromIndex)	Yields the index within this string of the last occurrence of the given substring.
int length()	Yields the length of this string.
boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)	Tests whether two string regions are equal, allowing you to ignore case.
boolean regionMatches(int toffset, String other, int ooffset, int len)	Tests whether two string regions are equal.
String replace(char oldChar, char newChar)	Yields a new string by replacing all occurrences of **oldChar** in this string with **newChar**.
boolean startsWith(String prefix)	Tests whether this string starts with the given prefix.
boolean startsWith(String prefix, int toffset)	Tests whether this string starts with the given prefix, beginning at the given index.
String substring(int beginIndex)	Yields a new string that's a substring of this string.
String substring(int beginIndex, int endIndex)	Yields a new string that's a substring of this string, allowing you to specify the end index.
char[] toCharArray()	Converts this string to a new character array.
String toLowerCase()	Converts all the characters in this String object to lowercase using the rules of the default locale, which is returned by **Locale.getDefault**.
String toLowerCase(Locale locale)	Converts all the characters in this **String** object to lowercase using the rules of the given locale.
String toString()	This object (which is already a string) is returned.
String toUpperCase()	Converts all the characters in this **String** object to uppercase using the rules of the default locale, which is returned by **Locale.getDefault**.
String toUpperCase(Locale locale)	Converts all the characters in this String object to uppercase using the rules of the given locale.
String trim()	Removes white space from both ends of this string.

Table 2.4 String class methods.

Method	Means
static String valueOf(boolean b)	Yields the string representation of the **boolean** argument.
static String valueOf(char c)	Yields the string representation of the char argument.
static String valueOf(char[] data)	Yields the string representation of the **char** array argument.
static String valueOf(char[ ] data, int offset, int count)	Yields the string representation of a specific subarray of the **char** array argument.
static String valueOf(double d)	Yields the string representation of a **double**.
static String valueOf(float f)	Yields the string representation of a **float**.
static String valueOf(int i)	Yields the string representation of an **int**.
static String valueOf(long l)	Yields the string representation of a **long**.
static String valueOf(Object obj)	Yields the string representation of an object.

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 I use one of the **String** class's constructors. In this case, I'm 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're using by the number and type of arguments you pass to it). You can even use the **String** class's **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);

        double double1 = 1.23456789;
        String s7 = String.valueOf(double1);
```

```
            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 to 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, I print out all the strings I've 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
1.23456789
```

Getting String Length

The Novice Programmer is breathless. "I've 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's **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** 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 I've 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's **concat** method to join two strings and create a new one.

How does that look in code? Here's an example where I 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've 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 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**:

```
public class app
{
    public static void main(String[] args)
    {
          String s1 = "I 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 I 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 = "I 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 7
The last occurrence of "draw" is at location 20
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 to lowercase, and you can use the **toUpperCase** method to convert it to 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!
```

Formatting Numbers in Strings

You can format numbers in strings using the **NumberFormat** class of the java.text package. This class supports the **format**, **setMinimumIntegerDigits**, set **MinimumFractionDigits**, **setMaximumInteger Digits**, and **setMaximumFractionDigits** methods. Here's an example, using the **setMaximumFraction Digits** method, in which I round off a **double** value as I 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've 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 in which I'm using the **StringBuffer** class's **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's constructors in Table 2.5 and its methods in Table 2.6.

I'll put the **StringBuffer** class to work in the next few topics.

Table 2.5 StringBuffer class constructors.

Constructors	Means
StringBuffer()	Constructs a string buffer with no characters in it and a capacity of 16 characters.
StringBuffer(int length)	Constructs a string buffer with no characters in it and a capacity as given by the length argument.
StringBuffer(String str)	Constructs a string buffer so that it represents the same sequence of characters as the argument string.

Table 2.6 StringBuffer class methods.

Method	Means
StringBuffer append(boolean b)	Appends the string representation of the **boolean** argument to the string buffer.
StringBuffer append(char c)	Appends the string representation of the **char** argument to the string buffer.
StringBuffer append(char[] str)	Appends the string representation of the **char** array argument to the string buffer.
StringBuffer append(char[] str, int offset, int len)	Appends the string representation of a subarray of the char array argument to the string buffer.

Table 2.6 StringBuffer class methods.

Method	Means
StringBuffer append(double d)	Appends the string representation of the **double** argument to the string buffer.
StringBuffer append(float f)	Appends the string representation of the **float** argument to the string buffer.
StringBuffer append(int i)	Appends the string representation of the **int** argument to the string buffer.
StringBuffer append(long l)	Appends the string representation of the **long** argument to the string buffer.
StringBuffer append(Object obj)	Appends the string representation of the **Object** argument to the string buffer.
StringBuffer append(String str)	Appends the string to the string buffer.
int capacity()	Yields the capacity of the string buffer.
char charAt(int index)	Yields the given character of the sequence represented by the string buffer, as indicated by the **index** argument.
StringBuffer delete(int start, int end)	Removes the characters in a substring of this string buffer.
StringBuffer deleteCharAt(int index)	Removes the character at the given position in this stringbuffer, shortening the string buffer by one character.
void ensureCapacity(int minimumCapacity)	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)	Characters are copied from this string buffer into the destination character array.
StringBuffer insert(int offset, boolean b)	Inserts the string representation of the **boolean** argument into the string buffer.
StringBuffer insert(int offset, char c)	Inserts the string representation of the **char** argument into the string buffer.
StringBuffer insert(int offset, char[ ] str)	Inserts the string representation of the **char** array argument into the string buffer.
StringBuffer insert(int index, char[ ] str, int offset, int len)	Inserts the string representation of a subarray of the **str** array argument into the string buffer.
StringBuffer insert(int offset, double d)	Inserts the string representation of the **double** argument into the string buffer.
StringBuffer insert(int offset, float f)	Inserts the string representation of the **float** argument into the string buffer.
StringBuffer insert(int offset, int i)	Inserts the string representation of the second **int** argument into the string buffer.
StringBuffer insert(int offset, long l)	Inserts the string representation of the **long** argument into the string buffer.
StringBuffer insert(int offset, Object obj)	Inserts the string representation of the **Object** argument into the string buffer.
StringBuffer insert(int offset, String str)	Inserts the string into the string buffer.

Table 2.6 StringBuffer class methods.

Method	Means
int length()	Yields the length (in characters) of this string buffer.
StringBuffer replace(int start, int end, String str)	Replaces the characters in a substring of the string buffer with the characters in the given string.
StringBuffer reverse()	The character sequence contained in this string buffer is replaced by the reverse of the sequence.
void setCharAt(int index, char ch)	The character at the given index of the string buffer is set to **ch**.
void setLength(int newLength)	Sets the length of the string buffer.
String substring(int start)	Yields a new string that contains a subsequence of characters currently contained in this string buffer. The substring begins at the given index.
String substring(int start, int end)	Yields a new string that contains a subsequence of characters currently contained in this string buffer.
String toString()	Converts to a string representing the data in this string buffer.

Creating String Buffers

You can create **StringBuffer** objects using the **StringBuffer** class's 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 initialise 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(10);
        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 String Buffer Lengths and Capacities

You can use the **StringBuffer** class's **length** method to find the lengths of the text in **StringBuffer** objects, and you can use the **capacity** method to find the amount of memory space allocated for that 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).

Here's an example that shows how to determine a string's length, how to determine a string's capacity (Java typically makes the capacity 16 characters longer than the length, to save time for future memory allocations), and how to 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 using the **setCharAt** method.

Here's an example in which I change the text "She had a wild look in her eyes". To "She had a mild look in her eyes". 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 String Buffers

"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**:

```
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 String Buffers

You can delete text in a **StringBuffer** object using the **delete** method and **deleteCharAt** methods. For example, here's how to change the text "I'm not having a good time." to "I'm having a good time." with **delete** (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("I'm not having a good time.");
        s1.delete(4, 8);
        System.out.println(s1);
    }
}
```

Here's the result:

```
C:\>java app
I'm having a good time.
```

Replacing Text in String Buffers

"I'm writing a text editor 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**; 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!
```

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 encapsulates 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 run, and will show the result as

```
C:\>java app
250     250
```

```
H    H
```

The following lines from the above 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 a 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. As only object references can be held by a collection, you are required to *box* primitive values into the corresponding **wrapper** class (for **int** it is **Integer**). 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 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 making it difficult to write and understand. Here, the autoboxing and unboxing comes in that is done automatically.

Lets 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 int1= intbox1;
          System.out.println("The value of intbox1 is "+ intbox1);
          System.out.println("The value of int1 is "+ int1);

          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 int2)
        {
            return int2;
        }
}
```

Let's discuss what is happening in the above program, from the code above we have,

```
public class app
{
    public static void main(String args[])
    {
        Integer intbox1,intbox2,intbox3;
        intbox1 =250;
        int int1= intbox1;
        System.out.println("The value of intbox1 is "+ intbox1);
        System.out.println("The value of int1 is "+ int1);
        .
        .
        .
    }
}
```

The line in the above program, is encapsulating a value within an object. This feature is known as *autoboxing* as, we have not defined the object explicitly using the **new** keyword:

```
intbox1 =250;
```

Again, the following line below is shown extracting the values. This feature is know as *Auto-unboxing*:

```
int int1= intbox1;
```

If we will 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** 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 int1= intbox1;
            System.out.println("The value of intbox1 is "+ intbox1);
            System.out.println("The value of int1 is "+ int1);

            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**, here 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 will 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 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 int1= intbox1;
            System.out.println("The value of intbox1 is "+ intbox1);
            System.out.println("The value of int1 is "+ int1);

            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 reference to an object. In such cases, the compiler will automatically convert those primitive values to the required class types where permissible. Lets take more lines of code,

```
public class app
{
    public static void main(String args[])
    {
        Integer intbox1,intbox2,intbox3;
        intbox1 =250;
        int int1= intbox1;
        System.out.println("The value of intbox1 is "+ intbox1);
        System.out.println("The value of int1 is "+ int1);

        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 int2)
    {
        return int2;
    }
}
```

Here we are defining a method **mthdbox** with **Integer** type parameter value **int1** is being passed, which will return the **int** value in another class **Autobox2**. So, in order to call the method **mthdbox**, note the expression **Integer intbox4 = Autobox2.mthdbox(500)** defined in the **main** class. The value can only be called by explicitly mentioning the class of the **function** from which it is called. So, the value of **int2** will be displayed as a result. The return statement will auto-unbox the expression.

Alternately, you can perform the autoboxing in methods using another way as shown below:

```
class app
{
    static int mthdbox(Integer int2)
    {
        return int2;
    }
    public static void main(String args[])
    {
        Integer intbox4 = mthdbox(500);
        System.out.println(intbox4);
    }
}
```

Here, there is no need to explicitly call the function using the class name as it itself contains the **main** function.

Varargs Fundamentals

The NP is tensed, "Is there an easy way by which I 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 is a new feature in Java 2, version 1.5. 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 below 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 the final parameter's type indicate that the final argument may be passed as an array or as a sequence of arguments. 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 **printf** 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 overloading gets called; but you can take advantage of them whenever the API offers them. Let us now understand the concept of varags through a small example given below:

```
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 above code is,

```
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 examples shown above 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()** function, method **mthdvar** has been called using three different arguments, in the first case we have not passed any arguments.

The contents of this array arguments can be fetched using the **for-each** statement. Let's briefly go through the **for-each** version of the **for** loop. The details of this you can get 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 app75
{
    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:

```
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
```

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()**. The decree of which function instance is meant is transparent to the user, removing 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 run-time 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 more hard to determine which overload 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

Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Average.Averg(Average.java:10)
        at Average.main(Average.java:15)
```

In this scenario, we need to be careful that the implementation handles an empty vararg list correctly. Here we could have avoided the issue either 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 an anytype 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:\>javac
C:\>java app
3
```

Fortunately, this issue doesn't come up too often.

Ambiguity in Varargs

However, varargs has it 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 be always faster on modern HotSpot than a single-element array (although naïve 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 that 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 (in bold) syntax introduces an inconsistency between the behaviour 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 that an equivalent syntax for initialising a local variable to multiple values like this: **Object... x = a, b;**. However, such syntax do 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 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 for. 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 popularise the use of the same.

Conditionals, and Loops

If you need an immediate solution to:	*See page:*
Operator Precedence	111
Incrementing and Decrementing: ++ and --	112
Unary **Not**: ~ And !	114
Multiplication and Division: * and /	114
Modulus: **%**	115
Addition and Subtraction: + and -	115
Shift Operators: >>, >>>, and <<	116
Relational Operators: >, >=, <, <=, ==, and !=	116
Bitwise and Bitwise Logical And, Xor, and Or: &, ^, and |	117
Logical && and ||	120
The **if-then-else** Operator: **?:**	121
Assignment Operators: = and **[operator]=**	122
Using the **Math** Class	124
Changes in the Math Class	125
Class StrictMath	125
Comparing Strings	126
The **if** Statement	127
The **else** Statement	128
Nested **If** Statements	129
Ther **If-else** Ladders	129
The **switch** Statement	130
The **while** Loop	132
The **do-while** Loop	134
The **for** Loop	136
The **For-Each** Loop	139

If you need an immediate solution to:	See page:
Supporting **for-each** in Your Own Class	142
A (Poor) Solution	142
Significance of For-each	143
Nested Loops	144
Using the **break** Statement	145
Using the **continue** Statement	146

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—adding, subtracting, dividing, multiplying, 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 I'll discuss all three of these programming topics here.

Operators

The most basic way to work with the data in a program is with the built-in Java operators. For example, say you've stored a value of 46 in one variable and a value of 4 in another. You can multiply those two values with the Java multiplication operator (*), as shown in this code:

```
public class app
{
    public static void main(String[] args)
    {
        int operand1 = 46, operand2 = 4, product;
        product = operand1 * operand2;
        System.out.println(operand1 + " * " + operand2 + " = " + product);
    }
}
```

Here's the result of this code:

```
C:\>java app
46 * 4 = 184
```

So, what operators are available in Java? 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, I'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. Java 2 version 1.5 also adds the* ***StrictMath*** *classes (you can find more about these classes at* ***http://java.sun.com/j2se/1.5/docs/api/index.html****), which 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.

For example, 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 I've 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's 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 I'll put them to work in this chapter, giving the Java syntax a thorough workout.

Loops

Loops are fundamental programming constructs that let you handle tasks by executing specific code repeatedly. For example, you might want to handle the items in a set of data by working with each item in succession, or you might want to keep performing a task until a particular condition becomes true. The basic loop statement involves the **for** statement, which lets you execute a block of code using a loop index. Each time through the loop, the loop index will have a different value, and you can use the loop index to specify a different data item in your data set, such as when you use the loop index as an index into an array.

Here's how you use a **for** loop in general (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 in curly braces):

```
for (initialization_expression; end_conditon; iteration_expression)
{
    statement
}
```

You can initialize a loop index in the initialization expression (in fact, you can use multiple loop indexes in a **for** loop), provide a test condition for ending the loop when that test condition becomes false in the end condition, and add some way of changing (usually incrementing) the loop index in the iteration expression.

Here's an example to make this clear. In this case, I'll use a **for** loop to sum up the grades of six students in an array and compute the average grade. Here's how this looks in code (note that I'm actually declaring and initializing the loop index to 0 in the initialization expression of the **for** loop, which Java allows, following the same custom in C++):

```
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);
    }
}
```

This code loops over all items in the **grades** array and adds them, leaving the result in the variable **sum**, which I then divide by the total number of entries in the array to find the average value. I loop over all elements using a loop index, which starts at 0 and is steadily incremented each time through the loop, up to the last item in the array. Here's the result of this code:

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

As you can see, the **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 using operators, conditional statements, and loops.

Immediate Solutions

Operator Precedence

"Hey", says the Novice Programmer, "Java's gone all wacky again. I tried adding 12 and 24 and then dividing the result by 6. The answer should have been 6, but Java said it's 16". "Probably an operator precedence problem", you say. "Let me check your code".

Java supports quite a number of operators, which might be a problem if you use a lot of them at once in a single statement. Which operator does Java execute first? For an example, take a look at the Novice Programmer's code, in which he tries to add 12 and 24 and then divide the sum by 6:

```
public class app
{
    public static void main(String[] args)
    {
        double value;
        value = 12 + 24 / 6;
        System.out.println("The value = " + value);
    }
}
```

Here's the actual result of this code:

```
C:\>java app
The value = 16.0
```

Clearly, there's something different going on from what the Novice Programmer expected. In fact, Java sets up a very clear precedence of operators, which means that if it finds two operators at the same level in a statement (that is, not enclosed in parentheses), it'll execute the operator with the higher precedence first. As it happens, the / operator has higher precedence than the + operator, so in the preceding expression, 24 is first divided by 6 and then the result is added to 12, which produces 16.

To specify to Java exactly the order in which you want operators to be evaluated, you can use parentheses to group the operations you want performed first. Here's how this would look for the previous example, where the parentheses around "12 + 24" make sure that this operation is performed first:

```
public class app
{
    public static void main(String[] args)
    {
        double value;
        value = (12 + 24) / 6;
        System.out.println("The value = " + value);
    }
}
```

Here's the result of this code:

```
C:\>java app
The value = 6.0
```

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, for example, that you can always use parentheses to set the execution order of operations in Java statements.

I'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			
()	[]	.	
++	--	~	!
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	[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 0, after you execute **value++**, **value** will hold 1. These operators were introduced in C to make incrementing and decrementing values, which are very common operations, easier. 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: ++ and -- can be either *postfix* operators (for example, **value++**) or *prefix* operators (for example, **++value**). When used as postfix operators, they're executed *after* the rest of the

statement, and when used as prefix operators, they're executed *before* the rest of the statement. This is something you have to watch out for. For example, take a look at the following code:

```
value2 = value1++;
```

In this case, when the statement is completed, **value2** will actually be left with the *original* value in **value1**, and the value in **value1** will be incremented. Here's an example showing how this works:

```
public class app
{
   public static void main(String[] args)
   {
       int value1 = 0, value2 = 0;

       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);

       int value3 = 0, value4 = 0;

       System.out.println();
       System.out.println("value3 = " + value3);
       System.out.println("value4 = " + value4);

       value4 = ++value3;

       System.out.println("After value4 = ++value3..");
       System.out.println("value3 = " + value3);
       System.out.println("value4 = " + value4);
   }
}
```

Here's the result of this code:

```
C:\>java app
value1 = 0
value2 = 0
After value2 = ++value1...
value1 = 1
value2 = 0
value3 = 0
value4 = 0
After value4 = ++value3...
value3 = 1
value4 = 1
```

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 I flip all the bits of the most positive short value, 32767, to find the most negative short value, and I also flip a boolean value from true to false:

```
public class app
{
    public static void main(String[] args)
    {
        short short1 = 32767;
        boolean boolean1 = true;
        System.out.println("Most negative short = " + ~short1);
        System.out.println("!true = " + !boolean1);
    }
}
```

Here's the result of this code:

```
C:\>java app
Most negative short = -32768
!true = false
```

If I had set **int1** 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 is 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 I use * and / on double values, and then I do the same thing on integer values. I 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: %

You use the modulus operator (%) to return the remainder of a division operation. For example, **10 / 3** equals 3 with a remainder of 1, so **10 % 3** equals 1. 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're converting to.

To see how this works, take a look at the solution "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're 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.

You use relational operators to create logical conditions that you can test with conditional statements such as the **if** statement. 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're 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 **And**ed 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 **And**ing 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're 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 I 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, I 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** (||) operators. Table 3.4 shows how these operators work. Here's how the 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. You use these operators to tie logical clauses together in a way that matches their names—use **And** when you want both logical clauses to be true and use Or when you only require one of two clauses to be true.

Here's an example taken from the previous topic in which I use &&:

```
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.
```

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 && and || operators also have another interesting property—they're *shortcircuit 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 I'm 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 I use the & operator but not when I 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've 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 I convert an integer between 0 and 15 into a hexadecimal digit using the ?: operator. This operator is perfect here, because I 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 I've 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, like this:

```
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 is 81". "I'll believe it when Java tells me so", says the NP.

You can use the **java.lang.Math** class to perform a great many math operations. For example, here's how to solve the Novice Programmer's problem 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));
    }
}
```

Here's the result:

```
C:\>java app
3 x 3 x 3 x 3 = 81.0
```

Here are the constants and methods of the **Math** class:

- **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

Related solution:	*Found on page:*
What Data Types Are Available	55

Changes in the Math Class

These are the various improvements to java.math of JDK 5.0:

- The **BigDecimal** class has added help for fixed-precision floating-point computation.
- The **Math** and **StrictMath** libraries include hyperbolic transcendental functions (sinh, cosh, tanh), cube root, base 10 logarithm, etc.
- The **Hexadecimal floating-point support** allows precise and predictable specification of particular floating-point values, hexadecimal notation can be used for floating-point literals and for string to floating-point conversion methods in **float** and **double**.

Class StrictMath

The class **StrictMath** contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

The Java **math** library is defined with respect to fdlibm version 5.3. Where fdlibm provides more than one definition for a function (such as acos), use the "IEEE 754 core function" version (residing in a file whose name begins with the letter c). The methods which require fdlibm semantics are **sin, cos, tan, asin, acos, atan, exp, log, log10, cbrt, atan2, pow, sinh, cosh, tanh, hypot, expm1**, and **log1p**.

- **static double cbrt (double a):** Returns the cube root of a double value.
- **static double cosh(double x):** Returns the hyperbolic cosine of a double value.
- **static double expm1(double x):** Returns ex -1.

- **static double hypot(double x , double y):** Returns sqrt(x2 +y2) without intermediate overflow or underflow.
- **static double log(double a):** Returns the natural logarithm (base e) of a double value.
- **static double log10(double a):** Returns the base 10 logarithm of a double value.
- **static double log1p(double x):** Returns the natural logarithm of the sum of the argument and 1.
- **static double rint(double a):** Returns the double value that is closest in value to the argument and is equal to a mathematical integer
- **static double signum(double d):** 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.i
- **static float signum(float f):** 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):** Returns the hyperbolic sine of a double value.
- **static double tanh(double x):** Returns the hyperbolic tangent of a double value
- **static double toDegrees(double angrad):** Converts an angle measured in radians to an approximately equivalent angle measured in degrees
- **static double toRadians(double angdeg):** Converts an angle measured in degrees to an approximately equivalent angle measured in radians.
- **static double ulp(double d):** Returns the size of an ulp of the argument.
- **static float ulp(float f):** Returns the size of an ulp of the argument

Comparing Strings

When you're 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, like this:

- **s1.equals(s2)** – Returns true if **s1** equals **s2**
- **s1.equalsIgnoreCase(s2)** – Returns true if **s1** equals **s2** (ignoring case)
- **s1.compareTo(s2)** – 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.

When you want to test conditions and execute code accordingly, it's time to use the Java conditional statements, such as the **if** statement. Here's how you use this statement in general:

```
if (condition) statement1;
else statement2;
```

Note that ***statement1*** and ***statement2*** can both be compound statements, which means they can be made up of a number of statements enclosed in curly braces. One way to get an absolute value, as the Novice Programmer was trying to do, is to start by checking whether the value is greater than zero and, if so, just print out the value itself. Here's how to make that test with an **if** statement:

```
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
```

In this case, the statement that's executed if the condition of the **if** statement is true is a single statement, but you can also execute multiple statements if you make them part of a compound statement in a code block, like this:

```
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

So far, the **if** statement in the absolute value example only displays an absolute value if the value, itself, is greater than zero. However, I can expand that **if** statement by adding an **else** clause, which is executed if the **if** statement's condition is false. Here's how this looks in code (note that I'm able to handle negative numbers as well as positive ones):

```
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** Statements

You can also nest if statements inside each other (that is, define them inside other if statements). Here's an example showing how this technique 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

It's possible to create an entire sequence of **if-else** statements, which is known as an *if-else ladder*. Here's an example showing how one works (in this case, I test the value in a string variable successively until I find a match to the current day of the week):

```
public class app
{
    public static void main(String[] args)
    {
        String day = "Wednesday";
        if(day == "Monday")
        System.out.println("It\'s Monday".);
        else if (day == "Tuesday")
            System.out.println("It\'s Tuesday".);
        else if (day == "Wednesday")
            System.out.println("It\'s Wednesday".);
```

```
        else if (day == "Thursday")
            System.out.println("It\'s Thursday".);
        else if (day == "Friday")
            System.out.println("It\'s Friday".);
        else if (day == "Saturday")
            System.out.println("It\'s Saturday".);
        else if (day == "Sunday")
            System.out.println("It\'s Sunday".);
    }
}
```

Here's the result of this code:

```
C:\>java app
It's Wednesday.
```

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

"Jeez", 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 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 I display the day of the week based on a numeric value using a **switch** statement:

```
public class app
{
    public static void main(String[] args)
    {
        int day = 3;
        switch(day)
        {
        case 0:
            System.out.println("It\'s Sunday".);
            break;
        case 1:
            System.out.println("It\'s Monday".);
            break;
        case 2:
            System.out.println("It\'s Tuesday".);
            break;
        case 3:
            System.out.println("It\'s Wednesday".);
            break;
        case 4:
            System.out.println("It\'s Thursday".);
            break;
        case 5:
            System.out.println("It\'s Friday".);
            break;
        default:
            System.out.println("It must be Saturday".);
        }
    }
}
```

Here's the result of this code:

```
C:\>java app
It's Wednesday.
```

You can even nest **switch** statements. Note that if you don't specify a **break** statement at the end of a **case** statement, execution will continue with the code in the next **case** statement. Sometimes, that's useful, such as when you want to execute the same code for multiple **case** test values, like this:

```
public class app
{
    public static void main(String[] args)
    {
        int temperature = 68;
        switch(temperature)
        {
        case 60:
        case 61:
```

```
            case 62:
            case 63:
            case 64:
            System.out.println("Too cold".);
            break;
            case 65:
            case 66:
            case 67:
            case 68:
            case 69:
            System.out.println("Cool".);
            break;
            case 70:
            case 71:
            case 72:
            case 73:
            case 74:
            case 75:
            System.out.println("Warm".);
            break;
            default:
            System.out.println("Probably too hot".);
            }
    }
}
```

Here's the result of this code:

```
C:\>java app
Cool.
```

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 keeps executing the statement in its body (which may be a compound statement, with a number of single statements inside curly braces) while a particular logical condition evaluates to true. Here's what a **while** loop looks like in general:

```
while(condition)
    statement
```

Note that if ***condition*** is not true, the body of the loop is not even executed once. Here's an example that puts the **while** loop to work; in this case, I display a value, successively subtract 1 from that value, and then display the result, as long as the result is positive. When the value becomes 0, the **while** loop stops, because the condition I've used (**value > 0**) has become false:

```
public class app
{
```

```
    public static void main(String[] args)
    {
        int value = 10;
        while (value > 0)
        {
            System.out.println("Current value = " + value--);
        }
    }
}
```

Here's what this **while** loop returns:

```
C:\JavaBB\Test>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 another **while** loop example in which I'm solving the Novice Programmer's problem and creating a program that will calculate factorials:

```
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
```

Here's a more advanced example. In this case, I'm converting a number to hexadecimal by successively stripping off hex digits with the modulus operator. Because the digits come off in reverse order, I'm 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, I pop them in another **while** loop to reverse the order of the digits and create the **StringBuffer** object to display:

```
import java.util.*;
public class app
{
   public static void main(String[] args)
   {
       int value = 32, temp = value;
       StringBuffer sb = new StringBuffer();
       Stack st = new Stack();
       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 + " yields 0x" + sb);
   }
}
```

Here's what this program's output looks like:

```
C:\>java app
Converting 32 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 just like a **while** loop, except that the test condition is evaluated at the end of the loop, not at the beginning. Here's what the **do-while** loop looks like (bear in mind that the statement can be a compound statement with a number of single statements inside curly braces):

```
do
   statement
while(condition);
```

The biggest reason to use a **do-while** loop instead of a **while** loop is when you need the body of the loop to be run at least once. For example, here's a case where the value I'm testing for isn't even available for testing until the end of the loop:

```
public class app
{
   public static void main(String[] args)
   {
      int values[] = {1, 2, 3, 0, 5}, test, index = 0;
      do
      {
         test = 5 * values[index++];
      } while (test < 15);
   }
}
```

On the other hand, there are times when you should use a **while** loop instead of a **do-while** loop—times when the body of the loop shouldn't even run once if the condition is not true. For example, take a look at the following code in which 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're 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're 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; end_conditon; iteration_expression)
{
    statement
}
```

You can initialize a loop index in the initialization expression (in fact, you can use multiple loop indexes in a **for** loop), provide a test condition for ending the loop when that test condition becomes false in the end condition, and provide some way of changing—usually incrementing—the loop index in the iteration expression.

Here's an example showing how to put the **for** loop to work (note that I start the loop index at 1 and end it when the loop index exceeds 10, which means that the loop body will execute exactly 10 times):

```
public class app
{
    public static void main(String[] args)
    {
        int loop_index;
        for (loop_index = 1; loop_index <= 10; loop_index++)
        {
            System.out.println("This is iteration number " + loop_index);
        }
    }
}
```

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
This is iteration number 6
This is iteration number 7
This is iteration number 8
This is iteration number 9
This is iteration number 10
```

Here's an example that you saw at the beginning of the chapter. This example finds the average student score by looping over all the scores and summing them (note that I'm 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's 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 I'm 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 any body at all—in fact, you can use a **null** statement. Here's an example in which I'm 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 in JDK 5.0 platform. 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 below 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 the JDK 5.0 compiler, and then pass in some random set of command-line arguments:

```
C:\javac Args.java
C:\java Args Hickery Dickery Dock
```

You'll get the provided arguments printed out to standard output, one per line:

```
Hickery
Dickery
Dock
```

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 above program by initializing the values in the program itself instead of giving values at run time.

Let us write another program using enumeration. This is created 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 contants. 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", assign it to the variable "element", and execute 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. Collections was finally added by J2SE 1.2. J2SE 5 has significantly increased the power and streamlined the use of collections framework.

Java's collection classes are designed to simplify the programming for applications that have to keep track of groups of objects. These classes are very powerful and surprisingly easy to use—at least the basics, anyway. The more advanced features of collection classes take some serious programming to get right, but for most applications, a few simple methods are all you need to use collection classes. I think it is very confusing for a newcomer to collections to have to wade through a class hierarchy that doesn't make sense until you know some of the details of how the basic classes work. You will learn more about the collections in Chapter 26. Let me give you about the brief introduction about **ArrayList** before going through the next program. An array list is the most basic type of Java collections. You can think of an array list as an array. It is similar to an array, but avoids many of the most common problems of working with arrays.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 below 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:

```
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 am 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 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 below 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:

```
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 stuff = new ArrayList();
        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:

```
42
-30
654.2
Name : Ethan Hawke
Name : Julie Delpy
Edited words : [Il ne lui faut que deux choses:, le, et, le, temps, - Alfred
de V
ignyl
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, I'm 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've 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 I'm 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 I 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++)
            {
                        inner: 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 I'm printing out reciprocals, and I want to avoid trying to print out the reciprocal of 0. If the current loop index equals 0, I 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
```

Object-Oriented Programming

If you need an immediate solution to:	See page:
Declaring and Creating Objects	156
Declaring and Defining Classes	159
Creating Instancre Variables	161
Setting Variable Access	162
Creating Class Variables	163
Creating Methods	165
Setting Method Access	166
Passing Parameters to Methods	167
Command-Line Arguments Passed to **main**	169
Returning Values from Methods	170
Creating Class Methods	171
Creating Data Access Methods	172
Creating Constructors	173
Passing Parameters to Constructors	174
A Full Class Example	175
Understanding Variable Scope	176
Using Recursion	177
Garbage Collection and Memory Management	178
Avoiding Circular References	179
Garbage Collection and the **finalize** Method	180
Overloading Methods	181
Overloading Constructors	182
Passing Objects to Methods	183

If you need an immediate solution to:	See page:
Passing Arrays to Methods	185
Using the **this** Keyword	186
Returning Objects from Methods	187
Returning Arrays from Methods	188
New Classes Added to Java Lang	189
The Process Builder Class	189
The String Builder Class	190
Catching an Exception	192
Nesting **try** Statements	198
Using **Finally** Clause	199
Throwing Exceptions	201
Creating a Custom Exception	203
Debugging Java Programs	204

In Depth

This chapter is all about a topic central to any Java program—object-oriented programming (OOP). I 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 semiautonomous, 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.

As I first discussed in Chapter 1, 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—Those 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 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 methods of base class in the derived class. All these constructs are important to object-oriented programming, and we will get into more details on each of them now.

NOTE: If you're 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 does not support 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 I create an object from the Java **String** class, passing the string I 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** 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 means 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 above-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, I use the **calculate** method to work with the two values in **operand1** and **operand2** and store the result of the calculation in **result**:

```
Calculator calc1 = new Calculator();
result = calc1.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've 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

Each type of exception is represented by a different exception class:

- **IllegalArgumentException:** Occurs when passing an incorrect argument to a method.
- **InputMismatchException:** Occurs when the console input doesn't match the data type expected by a method of the Scanner class.
- **ArithmeticException:** Occurs when an illegal type of arithmetic operation is performed.
- **IOException:** Occurs when the I/O encountered an unrecoverable I/O error.
- **ClassNotFoundException:** Occurs when a necessary class couldn't be found.

They are just some of the exceptions. You will find more 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 seen 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 inputstreamreader
{
    public static void main(String args[])
    {
       try
       {
           int character;
           InputStreamReader inputstreamreader = new
           InputStreamReader(System.in);
           while ((character = inputstreamreader.read()) != -1)
           {
```

```
                System.out.print((char) character);
            }
        }
        catch (IOException e)
        {
            System.out.println("Error!");
        }
    }
}
```

You will 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.

TIP: *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 know 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 dig into it until you've made it your own.

Immediate Solutions

Declaring and Creating Objects

The Novice Programmer (NP) appears, ready to discuss object-oriented programming. "I know all about objects now," the NP says, "only ..." "Only what?" you ask. "Only, 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 the 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 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, I'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, I can use the **new** operator, using this general form in which I'm 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()
```

You can also combine the declaration and creation steps into one step. Here's an example in which I'm 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 I'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 I 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 I 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 I've 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, I print out all the strings I'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.

Declaring and Defining Classes

The Novice Programmer is excited and says, "I've done it! I've created an object—it worked!" "Fine," you say approvingly, "now how about creating a class?" "Uh-oh," says the NP, "how does that work?"

There are two parts to 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. Here's the general form of a class declaration:

```
[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. Here's the general form of a class declaration and definition:

```
access class classname [extends ...] [implements ...]
{
   [access] [static] type instance_variable1;
   .
   .
   .
   [access] [static] type instance_variableN;
   [access] [static] type method1 (parameter_list)
   {
          .
          .
          .
   }
   .
   .
   .
   [access] [static] type methodN (parameter_list)
   {
          .
          .
          .
   }
}
```

The **static** keyword here 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, as you'll see in the next chapter.

An example will make this all clear. To start, I'll just create a very simple class named **printer** that defines one method, **print** (you first saw this example in Chapter 1). When I call the **print** method, it displays the message "Hello from Java!" on the console. Here's what this class looks like:

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

Now I can make use of the **print** method in other classes, as in this example, where I'm creating a new object of the **printer** class using the **new** operator and using 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();
    }
}
```

That's all it takes—now I put this code in a file (app.java), compile it, and then run it as follows:

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

Take a moment to study this example; note that I'm declaring and defining two classes, **printer** and **app** in the same file here. 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 here 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:

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

And here's the new app.java (note that I had to import the **printer** class to be able to use it; see Chapter 1 for more on importing classes):

```
import printer;
```

```
public class app
{
    public static void main(String[] args)
    {
            printer printer1 = new printer();
            printer1.print();
    }
}
```

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 store instance data in a class:

```
access class classname [extends ...] [implements ...]
{
    [access] type instance_variable1;
    .
    .
    .
    [access] type instance_variableN;
}
```

Here's an example in which I'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 I 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 darn 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 possible values for ***access*** are **public**, **private**, and **protected**. When you declare a class member **public**, it's accessible from anywhere in your program. If you declare it **private**, it's accessible only in the class it's a member of. If you declare it **protected**, it's available to the current class, other classes in the same package (you can group libraries of classes into Java packages; you've already seen some of the Java packages, such as java.lang, and you'll see how to create custom packages later in the book), and classes that are derived from that class. If you don't use an access specifier, the default access is that the class member is visible to the class it's a member of, to classes derived from that class in the same package, and to other classes in the same package. You can find the details in Table 4.1.

For example, if I wanted to make the instance variable **data_string** private to the **Data** class created in the previous topic, I 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 I try to access the **data_string** instance variable in another class, as I did previously in the **app** class, the Java compiler will object:

```
C:\>javac app.java -deprecation
app.java:12: Variable data_string in class Data not accessible from class
app.
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 instance_variable1;
    .
    .
    .
    [access] static type instance_variableN;
}
```

Here's an example in which I create a class named **data** with a class data variable named **intdata**:

```
class data
{
    public static int intdata = 0;
}
```

Now I can create two objects of the data class: **a** and **b**. When I set the **intdata** variable for **a** to 1, I 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)
    {
            data a;
            a = new data();
            System.out.println("The value of a.doubledintdata = " +
            a.doubledintdata);
    }
}
```

Here's the result of this code:

```
C:\>java app
The value of a.doubledintdata = 2
```


Related solution:	Found on page:
What Data Types Are Available?	55

Creating Methods

"OK," says the Novice Programmer, "I've 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."

We've been using methods ever since printing out our first message with **System.out.println**, so you're certainly familiar with the concept. 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] type method1 (parameter_list)
    {
            .
            .
            .
    }
    .
    .
    .
    [access] [static] type 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 body of the method—the code that will be executed when you call the method—is enclosed in a code block following the method's declaration.

Let's get to an example. In fact, you've already seen one earlier in this chapter—the **printer** class. In that example, I 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 I still use parentheses after the method's name—this is mandatory when you're 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 I'm 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.

Related solution:	*Found on page:*
Overriding Methods	225

Setting Method Access

"That darn 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 darn 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] type method1 (parameter_list)
    {
            .
            .
            .
    }
    .
    .
    .
    [access] [static] type methodN (parameter_list)
    {
            .
            .
            .
    }
}
```

The possible values for *access* are **public**, **private**, and **protected**. When you declare a class member **public**, it's accessible from anywhere in your program. If you declare it **private**, it's accessible only in the class it's a member of. If you declare it **protected**, it's available to the current class, to other classes in the same package (you can group libraries of classes into Java packages; you've already seen some of the Java packages, such as java.lang, and you'll see how to create custom packages later in the book), and to

classes that are derived from that class. If you don't use an access specifier, the default access is that the class member is visible to the class it's a member of, to classes derived from that class in the same package, and to other classes in the same package. You can find the details in Table 4.1.

Here's an example in which I 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] type 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 I pass the string to print to the **print** method. I 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 I 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)
    {
            (new 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 addem(int op1, int op2)
    {
            int result = op1 + op2;
            .
            .
            .
    }
```

```
}
```

You can call methods using literals, variables, arrays, or objects, like this:

```
calc.addem(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're 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**

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 Now is the time
```

In this case, the first element of the array passed to **main** will hold "Now", the second "is", the third "the", and the fourth "time". 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 I might put this application to work:

```
C:\>java app Now is the time
Command line arguments...
Argument 0 = Now
Argument 1 = is
Argument 2 = the
Argument 3 = time
```

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."

You use the **return** statement in a method to return a value from the method, and you indicate what the return type of the method is when you declare the method:

```
[access] [static] type method1 ([type parameter_name1 [, type
parameter_name1...]])
{
    .
    .
    .
}
```

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 **addem** that takes two integer parameters, adds them, and returns the result. Here's how I declare **addem**:

```
class calculator
{
    int addem(int op1, int op2)
    {
            .
            .
            .
    }
}
```

Here's how I return the sum of the values passed to **addem**, using the **return** statement:

```
class calculator
{
    int addem(int op1, int op2)
    {
            return op1 + op2;
    }
}
```

Here's how I put the **calculator** class to work in a program:

```
class calculator
{
    int addem(int op1, int op2)
    {
            return op1 + op2;
    }
}
```

```
public class app
{
    public static void main(String[] args)
    {
            calculator calc = new calculator();
            System.out.println("addem(2, 2) = " + calc.addem(2, 2));
    }
}
```

Here's the result of this application:

```
C:\>java app
addem(2, 2) = 4
```

Creating Class Methods

"Jeez," says the Novice Programmer, "I've created my new **calculator** class with a terrific method in it named **addem**, but why do I have to go to the trouble of creating an object of that class before I can use the **addem** method? Can't I just call that method directly?" "You can," you say, "if you make **addem** a *class* method instead of an *instance* method."

To make a method into a class method, use the **static** keyword:

```
class calculator
{
    static int addem(int op1, int op2)
    {
            return op1 + op2;
    }
}
```

Now in code, you can call the **addem** method directly 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("addem(2, 2) = " + calculator.addem(2, 2));
    }
}
```

Here's the result of this code:

```
C:\>java app
addem(2, 2) = 4
```

You can also use a class method the usual way—as a method of an object:

```
class calculator
{
    static int addem(int op1, int op2)
    {
```

```
            return op1 + op2;
    }
}

public class app
{
    public static void main(String[] args)
    {
            calculator calc = new calculator();
        System.out.println("addem(2, 2) = " + calc.addem(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 nonstatic methods from **main**? You do this as we've 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

"That darn Johnson," the Novice Programmer says, "is fiddling around inside my code's objects again. 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 darn Johnson!" the NP says.

You can restrict access to the data in your objects using data access methods, which must be called to fetch the data. Here's an example in which I've got a private String data member called **data_string**:

```
class data
{
    private String data_string = "Hello from Java!";
    .
    .
    .
}
```

I 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, I'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 I 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, which 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, without any access specifier or return type. Here's an example in which I 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 I 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. I'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 I 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

In this topic, I'll present an example using the concepts we've been discussing in this chapter so far. This example involves the simulation of a programming stack. You'll see stacks in more detail when we discuss Java collections, but the theory is a simple one—a programming stack works much like a stack of plates. When you put a plate on top of the stack, you're *pushing* an item onto the stack. When you take a plate off from the stack, you're *popping* an item off the stack. Note that the plates come off in reverse order—if you push plates 1, 2, and then 3, then when you pop the stack, plate 3 comes off first, followed by plates 2 and 1.

To use this **stack** class, you create an object of the class, passing an argument to the constructor indicating how big you want the stack (i.e., how many integers you want to be able to store on it). The constructor allocates the memory for the stack in an array named **stack_data**, and it sets up a stack pointer, **tack_ptr**, that points to the current top item on the stack (and this is actually the index I'll use with the **stack_data** array).

You can then use the stack's **push** method to push an item onto the stack, which stores a data item and increments the stack pointer to the next position in the stack array, or you can use the **pop** method to pop an item off the stack—the **pop** method returns the popped item and decrements the stack pointer. This application is called stacker.java; the test code at the end pushes 10 items onto the stack and then pops them off:

Here's what this program looks like at work:

```
C:\>java stacker
Pushing values now...
Pushed value--> 0
Pushed value--> 1
Pushed value--> 2
Pushed value--> 3
Pushed value--> 4
Pushed value--> 5
Pushed value--> 6
Pushed value--> 7
Pushed value--> 8
```

```
Pushed value--> 9

Popping values now...
Popped value--> 9
Popped value--> 8
Popped value--> 7
Popped value--> 6
Popped value--> 5
Popped value--> 4
Popped value--> 3
Popped value--> 2
Popped value--> 1
Popped value--> 0
```

Understanding Variable Scope

"Hmm," says the Novice Programmer, "I've 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 only visible in the method itself. 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 nonstatic 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 Class
{
   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)
    {
            Class c = new Class();
            c.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 I'll implement it here. To calculate the factorial of positive integer *n*, called "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 Novice Programmer 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 I'm 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're 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're 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 I 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're 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 **addem** 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 **addem** 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. I'll create the example the Novice Programmer was worried about here. First, I add a version of the **addem** method to the **calculator** class that will handle two operands:

```
class calculator
{
    int addem(int op1, int op2)
    {
            return op1 + op2;
    }
    .
    .
    .
}
```

Then I add another version of the same method that will take three operands:

```
class calculator
{
    int addem(int op1, int op2)
    {
            return op1 + op2;
    }
    int addem(int op1, int op2, int op3)
    {
            return op1 + op2 + op3;
    }
}
```

Now I can use both methods in code, like this:

```
public class app
{
    public static void main(String[] args)
    {
            calculator calc = new calculator();
            System.out.println("addem(2, 2) = " + calc.addem(2, 2));
            System.out.println("addem(2, 2, 2) = " + calc.addem(2, 2, 2));
```

```
    }
}
```

Here's the result of this program:

```
C:\>java app
addem(2, 2) = 4
addem(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 the details.

Overloading Constructors

"Wow," 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 works like overloading other methods (see the previous topic for the 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 that 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 I 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 I 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 I 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 only gets 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 I 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 I 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 Class
{
    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!");
            Class c = new Class();
            c.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, I'll pass an array to a method named **doubler** 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 I print out an array before and after the call to the **doubler** method):

```
class Calculate
{
    public void doubler(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[] = {1, 2, 3, 4, 5};
        Calculate c = new Calculate();
        System.out.println("Before the call to doubler...");
```

```
        for (int loop_index = 0; loop_index < array.length; loop_index++)
        {
            System.out.println("array[" + loop_index + "] = " +
            array[loop_index]);
        }
        c.doubler(array);
        System.out.println("After the call to doubler...");
        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 the call to doubler...
array[0] = 1
array[1] = 2
array[2] = 3
array[3] = 4
array[4] = 5
After the call to doubler...
array[0] = 2
array[1] = 4
array[2] = 6
array[3] = 8
array[4] = 10
```

Using the **this** Keyword

Java objects include a data member named **this** that's 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 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 I call the **getNewObject** method, it returns a new object of the **CreatedClass** class, and I 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 I create a class called **ArrayFactory** with a method named **getNewArray**. When I call **getNewArray**, it will return an array of integers. Note the return type I specify for **getNewArray** in the declaration: **int[]**. This indicates an integer array. Here's the code:

```
class ArrayFactory
{
    public int[] getNewArray()
    {
            int array[] = {1, 2, 3, 4, 5};
            return array;
    }
}
```

Here's how I 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] = 1
array[1] = 2
array[2] = 3
array[3] = 4
array[4] = 5
```

New Classes Added to Java Lang

"Fine, after going through all these facts there is one quarry, is there is any change in Java lang" the Novice Programmer asks. You say, "Yes my dear, there are lot of developments going on in the field of Java".

The Java lang has been enriched by two new classes: **"the process builder"** class and **"the string builder"** class.

The Process Builder 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 a new process instance. To create new subprocesses with identical/related attributes, the **start()** method can be invoked repeatedly from the same instance.

These process attributes are managed by each process builder in the following ways:

- *Command*—Command consists of a list of strings which make the external program file to be invoked and its arguments, if any, significant. The string lists representing a valid operating system command are system-dependent. For instance, for each conceptual argument to be an element in this list, it is a common practice, but we can find instances of operating systems where programs are expected to tokenize command line strings themselves—in such a scenario, a Java implementation might require commands to contain exactly 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 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 will get an immediate result of end of file, if we read from the **Process.getErrorStream()** returned stream.

The processes subsequently started by the object's start() method 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 modify the state of an object. For example, if the command attribute is set to an empty list it will not throw an exception unless start() is invoked.

***NOTE:** This class is not at all synchronized. If a **ProcessBuilder** instance is accessed concurrently by multiple threads, and at least one of the attributes is modified structurally by one of the threads, it must be synchronized externally.*

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 pbld1=new ProcessBuilder("notepad.exe", nfile.java");
    pbld1.start();
    }
}
```

In the above program, we have used an exception statement which you will study later in detail, but for information I am giving you the brief idea. An exception is an object that is created when an error occurs in a Java program and Java cannot automatically fix the error. An exception object contains the information about the error. Exceptions that occur belongs to different classes according to the occurrence of errors. A **throw** keyword used here simply lists the exceptions that the method might throw. The throw statement has the following basic format-

```
throw new exception-class();
```

When the program is executed, it will create a new file, or if it exists will open that, within the same directory. You will get a dialogue box as shown Figure 4.1.

Figure 4.1 Dialogbox Prompts to create a file.

The String Builder Class

The thread-safe **StringBuffer** class has a replacement in the form of **StringBuilder**. It does not have synchronized methods so it works much faster. So, if you are performing a number of **String** operations in a single thread, you will 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 string buffer (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 to a string by each of these methods and then appends or inserts the characters of that string to the string builder. 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 string builder object whose "start" are current contents, then the method that cause the string builder to contain "startle" is called **z.append("le")**, whereas the method that alter the string builder to contain "starlet" is called **z.insert(4,"le")**.

In general, if in an instance, a **StringBuilder** is referred by **sb**, then sb.append(x) has the same effect as **sb.insert(sb.length(), x)**. There is a capacity in every string builder. It is not necessary to allocate a new internal buffer, as long as the length of the character sequence contained in the string builder does not exceed the capacity. The string builder is automatically made larger, if the internal buffer overflows.

It is not safe to use instances of **StringBuilder** by multiple threads. It is recommended that **StringBuffer** be used if such synchronization is required.

***NOTE:** You can create new objects of this class from any of the four classes that implements the interface as "the string builder 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 using **StringBuilder**. The following code demonstrates these use, to give you more exposure to **StringBuilder**.

```
class app
{
    public static void main (String [] args)
    {
        System.out.println ("Concatenation: " +
        concat ("Hello", " ", "World"));
    }
    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.

```
Concatenation: Hello World
```

Catching an Exception

Let us take one more program of the **StringBuilder()** method.

```
import java.util.Scanner;
public class appp24
{
    static Scanner scan1 = new Scanner(System.in);
    public static void main(String[] args)
    {
```

```
            System.out.print("Enter a string: ");
            String str1 = scan1.nextLine();
            StringBuilder sbld1 = new StringBuilder(str1);
            int vowels = 0;
            for (int index_of_string = 0; index_of_string < str1.length();
index_of_string++)
            {
                  char chr1 = str1.charAt(index_of_string);
                  if (    (chr1 == 'A') || (chr1 == 'a')
                        || (chr1 == 'E') || (chr1 == 'e')
                        || (chr1 == 'I') || (chr1 == 'i')
                        || (chr1 == 'O') || (chr1 == 'o')
                        || (chr1 == 'U') || (chr1 == 'u') )
                  {
                        sbld1.setCharAt(index_of_string, '*');
                  }
            }
            System.out.println();
            System.out.println(str1);
            System.out.println(sbld1.toString());
      }
}
```

In the above code, we have used a special class called the **Scanner** class, which is explained in Chapter 12. But since I have used this class here, let me give you the brief introduction of it.

You can read input from either the keyboard or a text file by using **Scanner** class. This class provides simple and Sun-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 then converted into values of different types using the some methods discussed below.

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 will get the symbol '*', at the ('a','e','i','o','u') vowel's position. The output is shown below-

```
Enter a string: COMPUTER
COMPUTER
C*MP*T*R
```

Catching an Exception

"Darn," 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()	Constructs a new **Throwable** object.
Throwable(String message)	Constructs a new **Throwable** object with the indicated error message.

Table 4.3 Methods of the Throwable class.

Method	Does This
Throwable fillInStackTrace()	Fills in the execution stack trace.
String getLocalizedMessage()	Gets a localized description of this **Throwable** object.
String getMessage()	Gets the error message string of this **Throwable** object.
void printStackTrace()	Prints the stack trace to the standard error stream.
void printStackTrace(PrintStream s)	Prints the stack trace to the indicated print stream.
void printStackTrace(PrintWriter s)	Prints the stack trace to the indicated print writer.
String toString()	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 (Exception1Type e1)
{
    // Handle exception1 type
}
.
.
finally
{
    // Code to be executed before try block ends
}
```

Table 4.4 Constructors of the Exception class.

Copyrighted image

Throwable(String message)	Constructs a new Throwable object with the indicated error message.

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

Table 4.5 Some Java exception classes.

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 divzer
{
    public static void main(String[] args)
    {
        int d = 5;
        int z = 0;
        try
        {
            int quo = d/z;
        }
        catch (ArithmeticException e) { System.out.println ("Oops!!"); }
    }
}
```

Another example that uses a method to get a valid integer from the user is as follows:

```
import java.util.*;
public class chkInt
{
    static Scanner sc = new Scanner(System.in);
    public static void main(String[] args)
    {
        System.out.print("Enter an integer: ");
        int i = chkInt();
        System.out.println("You entered " + i);
    }
    public static int chkInt()
    {
        while (true)
        {
            try
            {
                return sc.nextInt();
            }
            catch (InputMismatchException e)
            {
                sc.next();
                System.out.print("That's not an integer. "
                    + "Try again: ");
            }
        }
```

```
    }
}
```

Here's the result (chkInt.java on the CD):

```
C:\> java chkInt
Enter an integer: 3.0
That's not an integer. Try again: 4.0
That's not an integer. Try again: 5.6
That's not an integer. Try again: 9
You entered 9
```

In this case, if the user enters a non-integer value, the catch block catches the error and forces the loop to repeat. But, if the user enters data that cannot be converted into an integer, the **nextInt**() method throws an **InputMismatchException**. Note that the program imports all the classes from the **java.util** package, including the Mismatch Exception class. The **nextInt**() method leaves the input value in the Scanner's input stream when an **InputMismatchException** is thrown. Then the **sc.next()** method in the catch block disposes off the user's invalid input. If this method (**sc.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, I'm catching an "array out of bounds" exception:

```
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 (excep.java on the CD accompanying this book), 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 **throw** keyword. Here's an example in which I indicate that the **doWork** 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 **throw** keyword in the method's definition (note that if you use the **throw** keyword like this, you don't need a **try/catch** block in the method's body):

```
class excep2
{
    public static void main(String args[])
    {
        try
        {
            doWork();
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
    static void doWork() throws ArithmeticException
    {
        int array[] = new int[100];
      array[100] = 100;
    }
}
```

Here's the result of this code (excep2.java on the CD). Note that the stack trace indicates that the exception occurred in the **doWork** method:

```
C:\>java excep2
Exception: 100
java.lang.ArrayIndexOutOfBoundsException: 100
    at excep2.doWork(excep2.java:12)
    at excep2.main(excep2.java:4)
```

Here is another example using the **FileNotFoundException**. A string is passed to the constructor that contains the path and the name of a file that exists on your computer. In case the file is not found, a **FileNotFoundException** is thrown. You will 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 will notice that you cannot compile this example successfully, as the compiler displays the following error:

```
C:\> java fileExcep
```

```
unreported exception java.io.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 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 (fileExcep.java on the CD), the following result gets displayed:

```
C:\>java fileExcep
File not found.
```

Nesting **try** Statements

You can nest **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:

```
class nested
{
    public static void main(String args[])
    {
        try
        {
            try
            {
                int c[] = {0, 1, 2, 3};
```

```
                c[4] = 4;
            }
            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 (nested.java on the CD):

```
C:\>java nested
Array index out of bounds: java.lang.ArrayIndexOutOfBoundsException: 4
```

Using 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 purpose is to clean up any mess that might be left behind by the exception, such as open files or database connections. Here's an example using Finally clause:

```
import java.util.*;
public class finExcep
{
    static Scanner sc = 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();
            int answer = divideTheseNumbers(i, j);
        }
        catch (Exception e)
        {
            System.out.println("...still it didn't work!");
        }
    }
    public static int getInt()
    {
```

```
            while (true)
                  try
                  {
                        return sc.nextInt();
                  }
                  catch (InputMismatchException e)
                  {
                        sc.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 (finExcep.java on the CD):

```
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

"That darn Johnson is at it again," says the Novice Programmer, "abusing my code." "Well," you say, "you can throw exceptions yourself in 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 I 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:

```
class thrower
{
    public static void main(String args[])
    {
            try
            {
                    doWork();
            }
            catch(ArithmeticException e)
            {
                    System.out.println("Caught in main" + e);
            }
    }
    static void doWork()
    {
            try
            {
                    throw new ArithmeticException("exception!");
            }
            catch(ArithmeticException e)
            {
                    System.out.println("Caught inside doWork " + e);
                    throw e;
            }
    }
}
```

Here's the output of this code (thrower.java on the CD):

```
C:\>java thrower
Caught inside doWork java.lang.ArithmeticException: exception!
Caught in mainjava.lang.ArithmeticException: exception!
```

Let us 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.*;
import java.lang.*;
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. This example is fileExcep1.java on the CD.

```
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 even throw a **FileNotFound** exception from the **main()** method also using a throws clause like this:

```
public static void main (String[] args) throws FileNotFoundException
{
    openFile("test1.bmp");
}
```

"Can I 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

"Darn," says the Novice Programmer, "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 NewException extends Exception
{
    int value;
    NewException(int v)
    {
        value = v;
    }
    public String toString()
    {
        return "NewException " + value;
    }
}
class customexception
{
    public static void main(String args[])
    {
        try
        {
            doWork(3);
            doWork(2);
            doWork(1);
            doWork(0);
        }
        catch (NewException e)
        {
            System.out.println("Exception: " + e);
        }
    }
    static void doWork(int value) throws NewException
    {
        if(value == 0)
        {
            throw new NewException(value);
        }
        else
        {
            System.out.println("No problem.");
        }
```

```
    }
}
```

Here's the output of this example (customexception.java on the CD):

```
C:\>java customexception
No problem.
No problem.
No problem.
Exception: NewException 0
```

Debugging Java Programs

"Uh-oh," says the Novice Programmer, "I've 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 I use the debugger on an application:

```
public class app
{
    public static void main(String[] args)
    {
        int a = 1, b = 2, c = 3, d = 4;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
    }
}
```

To start with, I 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 I start jdb just as I would 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 I do with the **help** command at the jdb prompt (see the highlighted line of code):

```
C:\>jdb app
Initializing jdb...
0xae:class(app)
> help
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> <expr> -- kill a thread with the given exception object
interrupt <thread> -- 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 <class id> -- break when specified exception thrown
ignore <class id> -- 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 methods [thread] -- trace method entry and exit
untrace methods [thread] -- stop tracing method entry and exit
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 id ... | "none"]
-- do not report step or method events for
specified
lasses
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
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
help (or ?) -- list commands
version -- print version information
exit (or quit) -- exit debugger
<class id>: full class name with package qualifiers or a
pattern 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
```

To debug, I first select a thread to work with. The **threads** command lists all available threads:

```
> threads
Group system:
1.(java.lang.Thread)0xb1 Sig dispatcher running
2.(java.lang.ref.Reference$ReferenceHandler)0xb2 Ref Handler cond. wait
3.(java.lang.ref.Finalizer$FinalizerThread)0xb3 Finalizer cond. wait
4.(java.lang.Thread)0xb4 Debugger agent running
5.(sun.tools.agent.Handler)0xb5 Breakpt handler cond. wait
6.(sun.tools.agent.StepHandler)0xb6 Step handler cond. wait
Group main:
7.(java.lang.Thread)0xb7 main cond. waiting
```

I'll work with the main thread here, thread 7, which I specify with the thread command:

```
> thread 7
main[1]
```

The jdb prompt changes to **main[1]**, which indicates the current method and line. I then set a breakpoint in the main method like this:

```
main[1] stop in app.main
Breakpoint set in app.main
```

Now I can run the code with the **run** command, which will run and stop immediately because of the breakpoint. To find out where I am, I use the **list** command to list the code:

```
main[1] run
run app
running ...
main[1]
Breakpoint hit: app.main (app:5)
main[1] list
```

```
1 public class app
2 {
3 public static void main(String[] args)
4 {
5 => int a = 1, b = 2, c = 3, d = 4;
6
7 System.out.println("a = " + a);
8 System.out.println("b = " + b);
9 System.out.println("c = " + c);
```

Now I can use the **next** command to single-step through the code, like this:

```
main[1] next
main[1] a = 1
Breakpoint hit: app.main (app:8)
main[1] next
main[1] b = 2
Breakpoint hit: app.main (app:9)
main[1] next
main[1] c = 3
```

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.

Inheritance, Inner Classes, and Interfaces

If you need an immediate solution to:	*See page:*
Creating a Subclass	215
Access Specifiers and Inheritance	216
Calling Superclass Constructors	218
Creating Multilevel Inheritance	222
Handling Multilevel Constructors	224
Overriding Methods	225
Accessing Overridden Members	226
Using Superclass Variables with Subclassed Objects	228
Dynamic Method Dispatch (Runtime Polymorphism)	230
Creating Abstract Classes	232
Stopping Overriding with **final**	233
Stopping Inheritance with **final**	235
Creating Constants with **final**	236
Is-a vs. Has-a Relationships	236
The Java **Object** Class	237
Using Interfaces for Multiple Inheritance	240
New Interfaces added to Java Lang	242
The Readable Interface	243
The Appendable Interface	243
The Iterable Interface	244
Creating Iterable Objects	247
Creating Inner Classes	250
Creating Anonymous Inner Classes	251

In Depth

This chapter is all about inheritance, a very important topic in Java programming. Using inheritance, you can derive one class, called the *derived class* or *subclass*, from another, called the *base class* or *superclass*. The idea here is that you add what you want to the new class to give it more customized functionality than the original class.

The previous chapter began our discussion of object-oriented programming, and as I 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 a **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 members 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 is, why is inheritance so important in Java?

Why Inheritance?

Java is truly an object-oriented language, and it relies on inheritance a great deal. The developers at Sun Microsystems have created huge packages—class libraries —full of 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 you first saw in Chapter 1 that creates a superclass based on the **Applet** class using the **extends** keyword (more on applets in the next chapter):

```
import java.applet.Applet;
import java.awt.*;
public class applet extends Applet
{
    public void paint(Graphics g)
    {
            g.drawString("Hello from Java!", 60, 100);
    }
}
```

Here's another example you saw in Chapter 1; in this case, I'm 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 from Java!", 60, 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.show();
    }
}
```

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. To create a standard applet, you can derive a class from the **java.applet.Applet** class, and to handle button clicks, you use another class, named **ActionListener**. Therefore, it looks as though you'll have to base your applet on both the **Applet** and **ActionListener** classes.

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, that means 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 we can use them in the chapters to come. There's one more topic I'll cover in this chapter—inner classes.

Why Inner Classes?

Java now lets you create classes within classes, and the enclosed class is called an *inner class*. I'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 loses 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:

```
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.show();
    }
}
```

We'll unravel this code in detail when working with events, and I'll get started on that process by introducing inner classes so that this code will make much more sense to you later. Now that you've gotten the concepts behind inheritance, interfaces, and inner classes down, it's time to turn to the "Immediate Solutions" section.

Immediate Solutions

Creating a Subclass

"OK," the Novice Programmer 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 using inheritance.

Suppose you have a class named **vehicle** that has one method, **start**, which you can use to start the vehicle and that prints out "Starting...":

```
class vehicle
{
    public void start()
    {
            System.out.println("Starting...");
    }
}
```

There are all kinds of vehicles, so if you want to specialize the **vehicle** class into a **car** class, you can use inheritance with the **extends** keyword. Here's how you declare **car** as a subclass of **vehicle**:

```
class car extends vehicle
{
    .
    .
    .
}
```

This syntax indicates that **car** is derived from the **vehicle** class, which means that in this case, **car** will inherit the **start** method from **vehicle**. You can also add your own data members and methods in subclasses. Here's an example in which I add a method named **drive** to the **car** class:

```
class car extends vehicle
{
    public void drive()
    {
            System.out.println("Driving...");
    }
}
```

Now I can access both the **start** method and the **drive** method in objects of the **car** class, as shown in this example:

```
public class app
{
    public static void main(String[] args)
    {
            System.out.println("Creating a car...");
```

```
            car c = new car();
            c.start();
            c.drive();
    }
}
```

Here's the output of the preceding code:

```
C:\>java app
Creating a car...
Starting...
Driving...
```

That's a basic subclassing example. There's a lot more to the process, however. 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 that's coming up in this chapter.

Access Specifiers and Inheritance

"Java's gone all wacky again," says the Novice Programmer. "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] type method1 (parameter_list)
    {
            .
            .
            .
    }
    .
    .
    .
   [access] [static] type methodN (parameter_list)
    {
            .
            .
            .
    }
}
```

The possible values for *access* are **public**, **private**, and **protected**. When you declare a class member **public**, it's accessible from anywhere in your program. If you declare it **private**, it's accessible only in the class it's a member of. If you declare it **protected**, it's available to the current class, to other classes in the same package, and to classes that are derived from that class. If you don't use an access specifier, the default access is that the class member is visible to the class it's a member of, to classes derived from that class in the same package, and to other classes in the same package. You can find the details in Table 5.1.

Table 5.1 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	X
Subclass in another package			x	X
Non-subclass in another package				X

For example, take a look at the Novice Programmer's code, where the start method is declared **private** but also accessed in **main**:

```
class vehicle
{
    private void start()
    {
            System.out.println("Starting...");
    }
}

class car extends vehicle
{
    public void drive()
    {
            System.out.println("Driving...");
    }
}

public class app
{
    public static void main(String[] args)
    {
            System.out.println("Creating a car...");
            car c = new car();
            c.start();
            c.drive();
    }
}
```

Because declaring a member using **private** restricts that member to its class, Java says it can't find the **start** method as used in **main**:

```
app.java:23: start() has private access in vehicle
```

```
c.start();
^
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 vehicle
{
    protected void start()
    {
            System.out.println("Starting...");
    }
}

class car extends vehicle
{
    public void drive()
    {
            System.out.println("Driving...");
    }
}

public class app
{
    public static void main(String[] args)
    {
            System.out.println("Creating a car...");
            car c = new car();
            c.start();
            c.drive();
    }
}
```

Here's the output of this application:

```
C:\>java app
Creating a car...
Starting...
Driving...
```

Calling Superclass Constructors

"OK," says the Novice Programmer, "I've 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 a that has a constructor with no parameters:

```
class a
{
    a()
    {
```

```
            System.out.println("In a\'s constructor...");
    }
}
```

Then you derive a subclass, **b**, from **a**:

```
class b extends a
{
}
```

Now when you create an object of class **b**, the constructor in class **a** is called automatically:

```
public class app
{
    public static void main(String[] args)
    {
            b obj = new b();
    }
}
```

Here's the result of the preceding code:

```
C:\>java app
In a's constructor...
```

Now suppose you add a constructor to class **b** that takes no parameters:

```
class a
{
    a()
    {
            System.out.println("In a\'s constructor...");
    }
}

class b extends a
{
    b()
    {
            System.out.println("In b\'s constructor...");
    }
}

public class app
{
    public static void main(String[] args)
    {
            b obj = new b();
    }
}
```

In this case, when you create an object of class **b**, the constructors from both **a** and **b** are called:

```
C:\>java app
```

```
In a's constructor...
In b's constructor...
```

Now suppose you change b's constructor so that it takes one parameter:

```
class a
{
    a()
    {
            System.out.println("In a\'s constructor...");
    }
}

class b extends a
{
    b(String s)
    {
            System.out.println("In b\'s String constructor...");
            System.out.println(s);
    }
}

public class app
{
    public static void main(String[] args)
    {
            b obj = new b("Hello from Java!");
    }
}
```

In this case, the constructors of both classes **a** and **b** are called:

```
C:\>java app
In a's constructor...
In b's String constructor...
Hello from Java!
```

A constructor with no parameters is called a *default constructor* for a class, because Java will call it automatically when you instantiate subclasses of that class, unless you make other arrangements. What does this mean? Suppose you add another constructor to **a** to handle strings as well (this is called *overloading*, and you'll see it later in this chapter):

```
class a
{
    a()
    {
            System.out.println("In a\'s constructor...");
    }
    a(String s)
    {
            System.out.println("In a\'s String constructor...");
            System.out.println(s);
    }
```

```
}
```

Now say that you want to call the constructor in class **a** with the **String** parameter instead of the default constructor. How do you do this? You do it by calling the **super** method in class **b**'s constructor, like this:

```
class a
{
   a()
   {
         System.out.println("In a\'s constructor...");
   }
   a(String s)
   {
         System.out.println("In a\'s String constructor...");
         System.out.println(s);
   }
}

class b extends a
{
   b(String s)
   {
         super(s);
         System.out.println("In b\'s String constructor...");
         System.out.println(s);
   }
}

public class app
{
   public static void main(String[] args)
   {
         b obj = new b("Hello from Java!");
   }
}
```

Now when you instantiate an object of class **b**, the constructor that takes a **String** parameter in class **a** is called, *not* the default constructor:

```
C:\>java app
In a's String constructor...
Hello from Java!
In b's String constructor...
Hello from Java!
```

Why is the constructor that takes a **String** parameter in class **a** called and not the default constructor in class **a**? The reason is simple—Java sees you're using the **super** method to call the superclass's method:

```
class b extends a
{
   b(String s)
   {
```

```
            super(s);
            System.out.println("In b\'s String constructor...");
            System.out.println(s);
    }
}
```

If you use **super** 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 Novice Programmer, "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, I created a class named **vehicle** and a subclass of **vehicle** named **car**:

```
class vehicle
{
    public void start()
    {
            System.out.println("Starting...");
    }
}

class car extends vehicle
{
    public void drive()
    {
            System.out.println("Driving...");
    }
}
```

That class hierarchy only included two levels—the superclass and the subclass— but things can go deeper. Say, for example, I have a new class, **aircraft**, that subclasses **vehicle** and add a method named **fly**:

```
class aircraft extends vehicle
{
    public void fly()
    {
            System.out.println("Flying...");
    }
}
```

There are all kinds of aircraft, and I'll derive two further classes from **aircraft**— **whirlybird**, which defines a new method named **whirl**, and **jet**, which defines a method named **zoom**:

```
class whirlybird extends aircraft
{
    public void whirl()
    {
```

```
            System.out.println("Whirling...");
    }
}

class jet extends aircraft
{
    public void zoom()
    {
            System.out.println("Zooming...");
    }
}
```

This is what the class hierarchy now looks like:

```
vehicle
|_____ car
|_____ aircraft
           |_______ whirlybird
           |_______ jet
```

Now I can instantiate objects of these classes, as in this example in which I'm creating objects of class **car** and **jet**:

```
public class app
{
    public static void main(String[] args)
    {
            System.out.println("Creating a car...");
            car c = new car();
            c.start();
            c.drive();
            System.out.println();
            System.out.println("Creating a jet...");
            jet j = new jet();
            j.start();
            j.fly();
            j.zoom();
    }
}
```

Here's what the output of this code looks like:

```
C:\>java app
Creating a car...
Starting...
Driving...
Creating a jet...
Starting...
Flying...
Zooming...
```

Handling Multilevel Constructors

"Hey!" the Novice Programmer says. "Has Java gone all wacky again?" you ask. "Yes," the NP says, "I've 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, I'll implement the Novice Programmer'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, I'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 I outlined in the topic "Calling Superclass Constructors," earlier in this chapter. However, all constructors in the subclassing chain must still be called in ascending order.

Overriding Methods

"Oh well," the Novice Programmer 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, I'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, however, 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, 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();
            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...
```


Related solution:	Found on page:
Creating Methods	165

Accessing Overridden Members

"Well," the Novice Programmer says, "I think I've run into a problem that no one else has ever encountered in Java." "Oh yes?" you ask. "Yes," the NP says, "I've 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();
        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 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 Novice Programmer says, "Run that by me again?" "Yes," you say in response to the PCC.

One interesting aspect of object-oriented programming (OOP) in Java—which I'll put to work in the next topic—is that you can assign a subclass object reference to a variable of a superclass type. Say, for example, that class **a** is the superclass of **b** and that you have a variable of class **a**. It turns out that you can assign object references of class **b** to that variable as well as object references of class **a**. Let's take a look at an example. Here, I'll use the multilevel class hierarchy you saw earlier in this chapter:

```
class vehicle
{
    public void start()
    {
            System.out.println("Starting...");
    }
}

class car extends vehicle
{
    public void drive()
    {
            System.out.println("Driving...");
    }
}

class aircraft extends vehicle
{
    public void fly()
    {
            System.out.println("Flying...");
    }
}

class whirlybird extends aircraft
{
    public void whirl()
    {
            System.out.println("Whirling...");
    }
```

```
}

class jet extends aircraft
{
    public void zoom()
    {
            System.out.println("Zooming...");
    }
}
```

For example, to create new objects of the **car** and **jet** classes, I can use this code:

```
public class app
{
    public static void main(String[] args)
    {
            System.out.println("Creating a car...");
            car c = new car();
            c.start();
            c.drive();
            System.out.println();
            System.out.println("Creating a jet...");
            jet j = new jet();
            j.start();
    }
}
```

However, I can also assign the new **jet** object to a variable of class **vehicle**, like this:

```
public class app
{
    public static void main(String[] args)
    {
            System.out.println("Creating a car...");
            car c = new car();
            c.start();
            c.drive();
            System.out.println();
            System.out.println("Creating a jet...");
            vehicle j = new jet();
            j.start();
            //j.fly();
            //j.zoom();
    }
}
```

Here's the output of this code:

```
C:\>java app
Creating a car...
Starting...
Driving...
```

```
Creating a jet...
Starting...
```

Note that I commented out the lines **j.fly()** and **j.zoom()** here because those methods are defined in the **aircraft** and **jet** classes, which are subclasses of **vehicle**, which means that those methods can't be used with a variable of class **vehicle**. In general, the object variable **a** will only permit access to those items that are members of its own class, not necessarily all the members of the object variable it happens to hold—in particular, you won't be able to access any members that are not members of **a**'s class.

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 Novice Programmer, "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, called *dynamic method dispatch* in Java, lets you wait until your program is actually running before specifying the type of object that will be in a particular object variable. This means, in the Novice Programmer'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 variable of a superclass type. You may be wondering why Java, which is very strict about typing, lets you do this. The answer is to support runtime polymorphism.

Here's an example to make this clearer. In this case, I'll create a superclass named **a**, a subclass of **a** named **b**, a subclass of **b** named **c**, and a subclass of **c** named **d**, each of which has a **print** method:

```
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...");
    }
}
```

Now I can create an object reference of each class type:

```
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();
            .
            .
            .
```

To show how runtime polymorphism works, I'll also create a variable named **aref** that holds an object reference to an object of class **a**:

```
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 I can place the object references to objects of all different classes in **aref**, and when I call the **print** method, the **print** method of the corresponding class will be called:

```
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;
            aref.print();
            aref = b1;
            aref.print();
            aref = c1;
            aref.print();
            aref = d1;
            aref.print();
    }
}
```

Here's the result of this code:

```
C:\>java app
Here's a...
Here's b...
Here's c...
Here's d...
```

Using runtime polymorphism, you can write 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: An object variable **a** will only permit access to those items that are members of its own class, not necessarily all the members of the object variable it happens to hold—in particular, you won't be able to access any members that are not members of **a**'s class.

Creating Abstract Classes

The Novice Programmer 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."

When writing classes, you may run across 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 customizes 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, I'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 I 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, I can make the method abstract, which means I must make the class, itself, abstract as well:

```
abstract class a
{
    abstract String getData();
    public void print()
    {
            System.out.println(getData());
    }
}
```

Now when I subclass **a**, I have to provide an implementation of **getData**, like this:

```
class b extends a
{
    String getData()
    {
            return "Hello from Java!";
    }
}
```

Here's how I 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 a great many methods in the Java packages themselves are abstract and therefore must be overridden.*

Stopping Overriding with **final**

The Novice Programmer 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**, like this:

```
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();
            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 -deprecation
```

```
app5.java:11: breathe() in fish cannot override breathe() in animal;
overridden
method is final
public void breathe()
^
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 -deprecation
app.java:9: Can't subclass final classes: class animal
class fish extends animal
^
1 error
```

Creating Constants with **final**

In the previous two topics, I showed two uses for **final**: to prevent method overriding and to prevent subclassing. There's another use for **final** in Java—you can use it to declare constants.

Here's an example in which I 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 -deprecation
app.java:7: Can't assign a value to a final variable: a
a = 6;
^
1 error
```

Is-a vs. Has-a Relationships

The Programming Correctness Czar 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 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're 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 Novice Programmer 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 knowing 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 5.2.

Here's an example in which I 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. I 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:

Table 5.2 The Java Object class methods.

Constructors	Means
protected Object clone()	Yields a copy of this object.
booleanequals(Object obj)	Indicates whether another object is equal to this one.
protected void finalize()	Called by the garbage collector on an object when garbage collection is about to dispose of the object.
Class getClass()	Yields the runtime class of an object.
int hashCode()	Yields a hashcode value for the object.
void notify()	Wakes up a single thread that's waiting on this object.
void notifyAll()	Wakes up all threads that are waiting on this object.
String toString()	Yields a string representation of the object.
void wait()	Makes the current thread wait until another thread invokes the **notify** method or the **notifyAll** method.
void wait(long timeout)	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)	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.

```
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, I 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 I 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 Novice Programmer, "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), like this:

```
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 I'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
{
    .
    .
    .
}
```

Because the interfaces declare but do 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, I 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 I defined like this:

```
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 -deprecation
clicker.java:5: class clicker must be declared abstract. It does not define
void actionPerformed(java.awt.event.ActionEvent) from interface
java.awt.event.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;
    .
    .
    .
```

New Interfaces added to Java Lang

"Fine, after going through all these facts there is one more query, is there any change in the Java lang" the Novice Programmer says. You say, "yes my dear there are lot of developments going on in the field of Java".

Three new interfaces, namely, "the **readable** interface", "the **appendable** interface" and "the **iterable** interface" have been added to Java lang during recent developments.

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 inheritance. 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. Though it need not be specified that they are abstract.

Table 5.3 Methods of Readable Interface.

Method	Means
int read(CharBuffer cb)	Attempts to read characters into the specified character buffer.

A **Readable** is a source of characters. The **CharBuffer** makes the characters available for the callers from the **Readable**. For instance, if **CharBuffer cb** is read, it tries to read characters into the specified character buffer. The only changes made are the results of a **put** operation, otherwise the buffer is used as a repository of characters as it is and no flipping or rewinding is performed for the buffer. So, the class **CharBuffer** should be inherited by **Readable** interface as follows:

```
public abstract class CharBuffer
extends Buffer
implements Comparable<CharBuffer>, Appendable, CharSequence, Readable
```

Considering another example, **InputStreamReader** in extended by **Reader** class, as follow:

```
public class InputStreamReader
extends Reader
```

An **InputStreamReader** is a bridge from byte streams to character streams—it reads bytes and decodes them into characters using a specified **charset**.

Now going to details of **Reader** class. This class in inherited by **Readable** interface, as shown below:

```
public abstract class Reader
extends Object
implements Readable, Closeable
```

As **InputStreamReader** class needs to use **read()** method, so it needs to be inherited by **Readable** class.

Readable is the interface implemented by objects type such as **BufferReader, CharBuffer, FileReader, InputStreamReader, Scanner** and a number of other readers; so use of this readable interface will be covered in chapter related to streams, i.e. Chapter 11—Working with stream: File and I/O handling.

The Appendable Interface

The interface to which **char** sequences and values can tagged on is an appendable interface. The appendable interface must be implemented for all classes for which instances are anticipated to receive formatted output from a formatter.

All characters that are required to be appended should be valid Unicode characters as described in Unicode Character Representation. Care must be taken to ensure that supplementary characters may be composed of 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 though 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 propagated to the invoker as this interface implemented by existing classes with different styles of error handling.

Since Java 5 the interface **Appendable**, consists of two methods which are also implemented by PrintStream:

```
interface java.IO. Appendable
```

Table 5.4 Methods of Appendable interface.

Method	Means
Appendable append(char c)	The indication C attaches to the current **Appendable** and returns the delivery of the current Object of type **Appendable**.
Appendable append(CharSequence csq)	The character sequence attaches at this **Appendable** and returns the delivery.

The interface is implemented beside **PrintStream** also by **CharBuffer** and all **Writer** classes, i.e. among other things by **BufferedWriter, CharArrayWriter, FileWriter, FilterWriter, OutputStreamWriter, StringWriter.** Use of this will be covered in Chapter 11.

From **PrintStream** and from **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;
}
```

Like **readable** interface, appendable will also be covered in Chapter 11 as this is also related to I/O streams and readers.

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. Implementing this interface allows an object to be the target of the "for-each" statement. Making a class iterable tells clients that they can iterate through its contents using a **for-each** loop. Sun has added a new interface to Java that allows you to mark your classes as iterable, this is the **java.lang.Iterable** interface:

Note: *Special care must be taken to note that Iterable is in* ***java.lang****, not* ***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 5.5 Methods of Iterable Interface.

Method	Means
Iterator<T> iterator()	Supplies an iterator, which iterates T over all elements of the type.

Many classes have already implemented this interface, so that with this extended interface the result quantities can be iterated. 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 forward and backward direction. This example uses an ArrayList object.

```
import java.util.*;
class Iter
{
    public static void main(String args[])
    {
        ArrayList<String> it = new ArrayList<String>();
        it.add("Z");
        it.add("Y");
        it.add("X");
        it.add("W");
        it.add("V");
        it.add("U");
        System.out.print("Contents of it are : ");
        Iterator<String> Sitr = it.iterator();
        while(Sitr.hasNext())
           {
                 String element = Sitr.next();
                 System.out.print(element + " ");
           }
        System.out.println();
        ListIterator<String> list_itr = it.listIterator();
        while(list_itr.hasNext())
           {
                 String element = list_itr.next();
                 list_itr.set(element + "+");
           }
        System.out.print("Contents of it after modification : ");
        Sitr = it.iterator();
        while(Sitr.hasNext())
           {
                 String element = Sitr.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:

```
Contents of it are : Z Y X W V U
Contents of it after modification : Z+ Y+ X+ W+ V+ U+
Contents of it in backward direction: U+ V+ W+ X+ Y+ Z+
```

Here's another example to implement a practical **Iterable**, in order to go over words of a sentence. Fundamental implementation of **Iterable** interface on array of characters serves the purpose of for-each loop and decides about whether characters of string can be selected or not.

We begin with the first part to be able the class **app**, which must implement iterable, in order on the right side of the point stand.

```
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:

```
At the beginning was the word - At the end the cliché
```

Let us see one more example related to **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**. The easiest way to provide an **Iterator** object to a client is to simply return 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();
    }
}
```

Note that the **Iterator**, the **List** reference, and the **ArrayList** are all bound to the **Product** type. Once the **java.lang.Iterable** interface has been implemented, the client code can use the **for-each** loop.

Here is an example:

```
Catalog catalog = new Catalog();

catalog.add(new Product("1", "pinto", new BigDecimal("4.99")));
catalog.add(new Product("2", "flounder", new BigDecimal("64.88")));
catalog.add(new Product("2", "cucumber", new BigDecimal("2.01")));

for (Product product: catalog)
product.discount(new BigDecimal("0.1"));

for (Product product: catalog)
System.out.println(product);
```

Creating Iterable Objects

For an object of a container class type to be usable with the collection based for loop, the class must fulfill one requirement. That is, it must implement the generic **Iterable<>** interface that is defined in the **java.lang** package.

```
public interface Iterable<T>
```

The **Iterable<>** interface is a generic type that declares a single method, **Iterator()**, that returns a reference of type **Iterator<>**, which is another generic interface type. Implementing this interface allows an object to be the target of the "**for-each**" statement.

- Iterator<T> **iterator()**—Returns an **Iterator** over a set of elements of type T.
- public interface **Iterator<E>**—An **Iterator** over a collection takes the place of Enumeration in the Java collections framework.

Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework. Take a look at this snippet using the **Iterator** interface:

```
Iterator iter = ...;
while (iter.hasNext())
{
   Object element = iter.next();                    // process element
}
```

In fact, anything that implements the **Iterable** interface can be used in a **for-each** loop, including your own classes. 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? Answer, the new **Iterable** interface indicates the presence of this method.

```
import java.util.ArrayList;
import java.util.Iterator;

public class app
{
    public static void main(String[] args)
   {
       double[] array =  {2.5, 5.2, 7.9, 4.3, 2.0, 4.1, 7.3, 0.1, 2.6};
       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("----------------------");
       for(int item: list)
       {
          System.out.println(item);
       }
       System.out.println("----------------------");
       MyIterableClass series = new MyIterableClass();
       for(double d: series)
       {
            System.out.println(d);
       }
```

```
    }
}
class MyIterableClass implements Iterable<Double>, Iterator<Double>
{
    static final double CLOSE_TO_ZERO = 0.0001;
    double value;
    double factor = -0.5;
    public MyIterableClass()
    {
            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 result = value;
            value = value*factor;
            return result;
    }
    public void remove()
    {
            throw new UnsupportedOperationException("There is nothing to
        remove");
    }
}
```

Here's the result of this code:

```
C:\>javac app.java
C:\>java app
2.5
5.2
7.9
4.3
2.0
4.1
7.3
0.1
2.6
----------------------
7
15
-67
----------------------
7
```

```
15
-67
---------------------
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 Novice Programmer, "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 but must instantiate and use an object instead, so they're not often used. *Inner classes*, on the other hand, are nonstatic nested classes, and they're 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 I 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*. I'll introduce them here, and we'll use them in chapter 7. An anonymous inner class is one that doesn't have a name, and you create it using this syntax:

```
new SuperType(constructor parameters)
{
   //methods and data
}
```

Here, ***SuperType*** is the name of a class or interface you're 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, I'll create an anonymous inner class inside a new class—class **a**. This anonymous inner class will subclass another class, class **b**, and define a method named **print**, which I 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 I 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!
```

AWT: Applets, Applications, and Event Handling

If you need an immediate solution to:	*See page:*
Using the Abstract Windowing Toolkit	259
Creating Applets	269
Using the <APPLET> HTML Tag	271
Handling Non-Java Browsers	274
Embedding <APPLET> Tags in Code	274
Using the init, start, stop, destroy, paint, and update Methods	275
Drawing Graphics in Applets	276
Reading Parameters in Applets	277
Using Java Consoles in Browsers	277
Adding Controls to Applets: Text Fields	278
Adding Controls to Applets: Buttons	280
Handling Events	281
Standard Event Handling	281
Using Delegated Classes	285
Using Action Commands	288
Handling Events the Old Way	289
Extending Components	289
Using Adapter Classes	291
Using Anonymous Inner Adapter Classes	293
Creating Windowed Applications	294
Exiting an Application When Its Window Is Closed	299
Applications You Can Run as Applets	300
Setting Applet Security Policies	301

In Depth

We've 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, I'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. Recently 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 now called *heavyweight components* because of their significant use of system resources, and the Swing components are called *lightweight components*, because they're 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 in (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—I'll spend this and the next few chapters on AWT. I'll start with an overview of AWT itself.

The Abstract Windowing Toolkit

It's no exaggeration to say that the Abstract Windowing Toolkit 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**—Creates an applet.
- **Button**—Creates a button.
- **Canvas**—Creates a canvas you can draw in.
- **Checkbox**—Creates a checkbox.
- **Choice**—Creates a choice control.
- **Label**—Creates a label.
- **Menu**—Creates a menu.
- **ComboBox**—Creates a combo box.
- **List**—Creates a list control.
- **Frame**—Creates a frame for windowed applications.
- **Dialog**—Creates a dialog box.
- **Panel**—Creates a panel that can contain other controls.

- **PopupMenu**—Creates a pop-up menu.
- **RadioButton**—Creates a radio button.
- **ScrollBar**—Creates a scrollbar.
- **ScrollPane**—Creates a scrollable surface.
- **TextArea**—Creates a two-dimensional text control.
- **TextField**—Creates a one-dimensional text field (called a *text box* in other languages).
- **TextPane**—Creates a text surface.
- **Window**—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 just a class file that's specially written to display graphics in a Web browser; you embed applets in Web pages using the HTML **<APPLET>** tag. When run in a Web page, Java applets are downloaded automatically and run by the Web browser; they're displayed in the space you've allocated for them in the page. They can do anything from working with graphics, to displaying animation, to handling controls (like the ones we'll work on in this chapter), text fields, and buttons. Using applets makes your Web pages *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 AWT **Component** class. Here's an example you've seen before, and I'll go through it again in this chapter. This example just displays the text "Hello from Java!" in a Web page:

```
import java.applet.Applet;
import java.awt.*;
public class applet extends Applet
{
    public void paint(Graphics g)
    {
            g.drawString("Hello from Java!", 60, 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 applet .class file (for example, in Unix, you might give the applet the permission setti g 644, which lets anyone read the file).

***TIP:** Unix file permissions make up three octal digits corresponding to, in order, 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 = "newApplet.class"
    WIDTH = 300
    HEIGHT = 200
>
</APPLET>
</CENTER>
</BODY>
</HTML>
```

In this case, I've set up a centered 300-by-200-pixel space in a Web page in which to display the applet, and I told the Web browser to download the newApplet.class file and run it. I'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 VERY IMPORTANT thing to know about now that we're 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 2 version 1.5 will *not* work in the latest browsers, even Microsoft Internet Explorer and Netscape Navigator (you can always use the Sun appletviewer, of course). I get dozens of letters asking why the examples in this book won't work—weren'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 it lets you run applets using the latest Java version. I'll cover using the plug-in in this chapter—see the solution "The Java Plug-In: Running the Latest Applets in Your Browser." 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 "Hello from Java!" in a frame window:

```
import java.awt.*;
import java.awt.event.*;
class AppFrame extends Frame
{
    public void paint(Graphics g)
    {
            g.drawString("Hello from Java!", 60, 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.show();
    }
}
```

I'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 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, I'll also introduce two basic controls in this chapter—buttons and text fields. The user can use the 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, I'll create programs in this chapter that 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 Windowing Toolkit

"OK," says the Novice Programmer, "I'm ready to start working with the Abstract Windowing Toolkit. 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 huge number of methods, many of which you'll see in this and the upcoming chapter. I've listed them all in Table 6.1 for reference. This is a long table, but it's worth glancing through and coming back to later for reference.

Table 6.1 Methods of the AWT Component class.

Method	Does This
boolean action(Event evt, Object what)	Deprecated. You should register the component with an **ActionListener**.
void add(PopupMenu popup)	Adds the pop-up menu to the component with an **ActionListener**. component.
void addComponentListener(ComponentListener l)	Adds the component listener to receive component events.
void addFocusListener(FocusListener l)	Adds the focus listener to receive focus events.
void AddHierarchyBoundsListener(HierarchyBoundsListener l)	Adds the hierarchy bounds listener to get hierarchy bounds events from this component when the hierarchy to which this container belongs changes.
void addHierarchyListener(HierarchyListener l)	Adds the hierarchy listener to get hierarchy changed events from this component when the hierarchy to which this container belongs changes.
void addInputMethodListener(InputMethodListener l)	Adds the input method listener to receive input method events.
void addKeyListener(KeyListener l)	Adds the key listener to receive key events.
void addMouseListener(MouseListener l)	Adds the mouse listener to receive mouse events.
void addMouseMotionListener(MouseMotionListener l)	Adds the mouse motion listener to receive mouse motion events.
void addMouseWheelListener(MouseWheelListener l)	Adds the mouse wheel listener to receive mouse wheel events.
void addNotify()	Makes a component "displayable" by connecting it to a native screen resource.
void addPropertyChangeListener(PropertyChangeListen	Adds a PropertyChangeListener.

Table 6.1 Methods of the AWT Component class.

Method	Does This
er listener)	
void addPropertyChangeListener(String propertyName, PropertyChangeListener listener)	Adds a **PropertyChangeListener** for a property.
void applyComponentOrientation(ComponentOrientation orientation)	Sets the component orientation property of this component and all component contained within it.
boolean areFocusTraversalKeysSet(int id)	Returns whether the focus traversal keys are defined.
rectangle bounds()	Deprecated. Replaced by **getBounds()**.
int checkImage(Image image, ImageObserver observer)	Gets the status of the screen representation of the image.
int checkImage(Image image, int width, int height ImageObserver observer)	Gets the status of the screen representtation of the image.
protected AWTEvent coalesceEvents(AWTEvent existingEvent, AWTEvent newEvent)	Coalesces an event being posted with another event.
boolean contains(int x, int y)	Checks whether this component contains the indicated point.
boolean contains(Point p)	Checks whether this component contains the indicated point.
Image createImage(ImageProducer producer)	Creates an image from the indicated image producer.
Image createImage(int width, int height)	Creates an offscreen image to be used for double buffering.
VolatileImage createVolatileImage(int width, int height)	Creates a volatile off screen drawable image.
VolatileImage createVolatileImage(int width, int height, Image Capabilities caps)	Creates a volatile off screen drawable image with the given capabilities.
void deliverEvent(Event e)	Deprecated. Replaced by dispatchEvent(AWTEvent e).
void disable()	Deprecated. Replaced by **setEnabled(boolean)**.
protected void disableEvents(long eventsToDisable)	Disables the events defined by the indicated event mask parameter from being sent to this component.
void dispatchEvent(AWTEvent e)	Dispatches an event to this component or one of its subcomponents.
void doLayout()	Makes the layout manager lay out this component.
void enable()	Deprecated. Replaced by **setEnabled(boolean)**.
void enable(boolean b)	Deprecated. Replaced by **setEnabled(boolean)**.
protected void enableEvents(long eventsToEnable)	Enables the events defined by the indicated event mask parameter to be sent to this component.
void enableInputMethods(boolean enable)	Enables or disables input method support.
protected void firePropertyChange(String propertyName, Object oldValue, Object newValue)	Sets up support for reporting bound property changes.

Table 6.1 Methods of the AWT Component class.

Method	Does This
void firePropertyChange(String propertyName, byte oldValue, byte newValue)	Reports a bound property change.
void firePropertyChange(String propertyName, char oldValue, char newValue)	Reports a bound property change.
void firePropertyChange(String propertyName, double oldValue, double newValue)	Reports a bound property change.
void firePropertyChange(String propertyName, float oldValue, float newValue)	Reports a bound property change.
protected void firePropertyChange(String propertyName, int oldValue, int newValue)	Support for reporting bound property changes for integer property.
void firePropertyChange(String propertyName, long oldValue, long newValue)	Reports a bound property change.
void firePropertyChange(String propertyName, short oldValue, short newValue)	Reports a bound property change.
float getAlignmentX()	Gets the alignment along the x axis.
float getAlignmentY()	Gets the alignment along the y axis.
Color getBackground()	Gets the background color of this component.
Rectangle getBounds()	Ties the bounds of this component in a **Rectangle** object.
Rectangle getBounds(Rectangle rv)	Stores the bounds of this component into **rv** and **return rv**.
ColorModel getColorModel()	Gets the instance of **ColorModel** used to display the component.
Component getComponentAt(int x, int y)	Determines whether this component or one of its immediate subcomponents contains the (x, y) location and gets the containing component.
Component getComponentAt(Point p)	Gets the component or subcomponent that contains the indicated point.
Component Listener[] getComponentListeners()	Returns an array of all the component listeners.
ComponentOrientation getComponentOrientation()	Retrieves the language-sensitive orientation to be used to order the elements or text within this component.
Cursor getCursor()	Gets the cursor set in the component.
DropTarget getDropTarget()	Gets the drop target connected to this component.
Container getFocusCycleRootAncestor()	Returns the container which is the focus cycle root of the component.
Font getFont()	Gets the font of this component.
FontMetrics getFontMetrics(Font font)	Gets the font metrics for the indicated font.
Color getForeground()	Gets the foreground color of this component.
Graphics getGraphics()	Creates a graphics context for this component.

Table 6.1 Methods of the AWT Component class.

Method	Does This
GraphicsConfiguration getGraphicsConfiguration()	Gets the **GraphicsConfiguration** associated with this component.
int getHeight()	Returns the current height of this component.
HierarchyBoundsListeners[] get hierarchialyBoundsListeners()	Returns an array of all the hierarchy bound listeners.
InputContext getInputContext()	Gets the input context used by this component.
InputMethodRequests getInputMethodRequests()	Gets the input method request handler.
EventListener [] getListeners(Class listenerType)	Returns an array of all the listeners that were added to the component with **addXXXListener()**, where **XXX** is the name of the **listenerType** argument.
Locale getLocale()	Gets the locale of this component.
Point getLocation()	Gets the location of this component in the form of a point specifying the component's top-left corner.
Point getLocation(Point rv)	Stores the x,y origin of this component into **rv** and **return rv**.
Point getLocationOnScreen()	Gets the location of this component in the form of a point specifying the component's top-left corner.
MouseListener[] getMouseListeners()	Returns an array of all the mouse listener.
Dimension getMaximumSize()	Gets the maximum size of this component.
Dimension getMinimumSize()	Gets the minimum size of this component.
String getName()	Gets the name of the component.
Container getParent()	Gets the parent of this component.
java.awt.peer.ComponentPeer getPeer()	Deprecated. Replaced by **Boolean IsDisplayable()**.
PropertyChangeListener[] get PropertChangeListeners()	Returns an array of all the property change listener.
PropertyChangeListener[] get PropertChangeListeners (string property name)	Returns an array of all the listeners which have been associated with the name property.
Dimension getPreferredSize()	Gets the preferred size of this component.
Dimension getSize()	Gets the size of this component in a **Dimension** object.
Dimension getSize(Dimension rv)	Stores the width/height of this component into **rv** and **Return rv**.
Toolkit getToolkit()	Gets the toolkit of this component.
Object getTreeLock()	Gets the locking object for the AWT component tree.
Int getWidth()	Gets the current width of this component.
int getX()	Gets the current x coordinate of the component's origin.
Int getY()	Gets the current y coordinate of the component's origin.
boolean gotFocus(Event evt, Object what)	Deprecated. Replaced by

Table 6.1 Methods of the AWT Component class.

Method	Does This
	processFocusEvent(FocusEvent).
boolean handleEvent(Event evt)	Deprecated. Replaced by **processEvent(AWTEvent)**.
boolean hasFocus()	Returns true if this component has the keyboard focus.
void hide()	Deprecated. Replaced by **setVisible(boolean)**.
boolean imageUpdate(Image img, int flags, int x, int y, int w, int h)	Repaints the component when the image has changed.
boolean inside(int x, int y)	Deprecated. Replaced by **contains(int, int)**.
void invalidate()	Invalidates the component.
boolean isDisplayable()	Determines whether the component can be displayed.
boolean isDoubleBuffered()	True if this component is painted to an offscreen image that's copied to the screen later.
boolean isEnabled()	Indicates whether this component is enabled.
boolean isFocusTraversable()	Indicates whether this component can be traversed using Tab or Shift+Tab.
boolean isFocusCycleRoot(Container cContainer)	Returns whether the specified container is the focus cycle root of the component focus traversal cycle.
boolean isFocusOwner()	Returns true if this component is the focus owner.
boolean isLightweight()	Indicates whether the component is lightweight.
boolean isOpaque()	True if this component is completely opaque; false by default.
boolean isShowing()	Determines whether this component is visible onscreen.
boolean isValid()	Determines whether this component is valid.
boolean isVisible()	Determines whether this component should be visible when its parent is visible.
boolean keyDown(Event evt, int key)	Deprecated. Replaced by **processKeyEvent(KeyEvent)**.
boolean keyUp(Event evt, int key)	Deprecated. Replaced by **processKeyEvent(KeyEvent)**.
void layout()	Deprecated. Replaced by **doLayout()**.
void list()	Prints a listing of this component to **System.out**.
void list(PrintStream out)	Prints a listing of this component to the indicated output stream.
void list(PrintStream out, int indent)	Prints out a list, starting at the indicated indention, to the indicated print stream.
void list(PrintWriter out)	Prints a listing to the indicated print writer.
void list(PrintWriter out, int indent)	Prints out a list, starting at the indicated indention, to the indicated print writer.

Table 6.1 Methods of the AWT Component class.

Method	Does This
Component locate(int x, int y)	Deprecated. Replaced by **getComponentAt(int, int)**.
Point location()	Deprecated. Replaced by **getLocation()**.
boolean lostFocus(Event evt, Object what)	Deprecated. Replaced by **processFocusEvent(FocusEvent)**.
Dimension minimumSize()	Deprecated. Replaced by **getMinimumSize()**.
boolean mouseDown(Event evt, int x, int y)	Deprecated. Replaced by **processMouseEvent(MouseEvent)**.
boolean mouseDrag(Event evt, int x, int y)	Deprecated. Replaced by **processMouseMotionEvent(MouseEvent)**.
boolean mouseEnter(Event evt, int x, int y)	Deprecated. Replaced by **processMouseEvent(MouseEvent)**.
boolean mouseExit(Event evt, int x, int y)	Deprecated. Replaced by **processMouseEvent(MouseEvent)**.
boolean mouseMove(Event evt, int x, int y)	Deprecated. Replaced by **processMouseMotionEvent(MouseEvent)**.
boolean mouseUp(Event evt, int x, int y)	Deprecated. Replaced by **processMouseEvent(MouseEvent)**.
void move(int x, int y)	Deprecated. Replaced by **setLocation(int, int)**.
void nextFocus()	Deprecated. Replaced by **transferFocus()**.
void paint(Graphics g)	Paints this component.
void paintAll(Graphics g)	Paints this component and all subcomponents.
protected String paramString()	Gets a string representing the state of this component.
boolean postEvent(Event e)	Deprecated. Replaced by **dispatchEvent(AWTEvent)**.
Dimension preferredSize()	Deprecated. Replaced by **getPreferredSize()**.
boolean prepareImage(Image image, ImageObserver observer)	Prepares an image for rendering on this component.
boolean prepareImage(Image image, int width, int height, ImageObserver observer)	Prepares an image for rendering on this component at the indicated width and height.
void print(Graphics g)	Prints the component.
void printAll(Graphics g)	Prints the component and all its subcomponents.
protected void processComponentEvent(ComponentEvent e)	Processes component events occurring in this component by sending them to any registered **ComponentListeners**.
protected void processEvent(AWTEvent e)	Processes events occurring in this component.
protected void processFocusEvent(FocusEvent e)	Processes focus events occurring in this component by dispatching them to any registered **FocusListener** objects.

Java 2

(JDK 5 Edition)

AWT, Swing, Generics, XML, Sound, Animation, JDBC, Servlets, RMI, Threading, Sockets, Networking and Java Beans

Programming Black Book™

The Standard in Java Coverage—More Complete and Accessible Than Ever Before!

Written by an award-winning author, Java 2 Black Book contains nearly 700 Java examples covering all aspects of the Java language This revised edition of the best-selling Java Black Book reflects changes available in the latest syntax version of Java, including: Generics; drag and drop; security enhancements; the new applet deployment enhancements; Networking; Servlets; XML; Sound and Animation; database handling; Java Naming and Directory Interface; and the new Enumeration. Each topic is illustrated with a full, working example, rather than partial code. This book also features additional coverage of Java and XML. Plus, the CD-ROM contains other useful tools and codes!

This book will help you:

- Master the full Java 2
- Understand the Abstract Windowing Toolkit (AWT)
- Conquer the Java Swing user interface
- Use Java to handle XML (both DOM and SAX parsing)
- Employ Networking and Sockets on the Internet
- Utilize Java Database Connectivity
- Form multithreaded applications
- Handle advanced security issues
- Generate Java Beans and Servlets
- Create both TCP clients and servers
- Submit HTML form from Java
- Build packages, interfaces and JAR files

The CD-ROM for this book is packed with tools you can use right away. Look inside the back cover for more details.

ISBN 10: 81-7722-455-X
ISBN 13: 978-81-7722-455-3

CATEGORY:
Programming/Java

USER LEVEL:
Intermediate to Advanced

INDIAN PRICE: Rs. 499/- with CD
International Edition Rs. 2750/-

Authorized for Sale in the Indian Sub-continent Only

New Features

- **Generics**
- **Metadata**
- **Enumeration**
- **Autoboxing and Unboxing**

What Is In This Book For You!

- Complete coverage of Java 2 (JDK 1.5)
- Master Java Internet Handling
- Understand Java Swing
- Learn to manage XML with Java
- Discover the world of RMI
- Delve into the world of Sound and Animation
- Walk through Servlets and Java Beans
- Get skilled with Database Handling
- Uncover Threading
- Get in-depth knowledge on Sockets in Networking
- Understand Serialization
- Grasp Synth, a skinnable look and feel

About the Author

Steven Holzner is an award-winning author who has written many books on Java, his favorite topic, but this is his most comprehensive book to date. He's written a total of 67 books on computing, and his books have been translated into 16 different languages. A former PC Magazine contributing editor, he is a graduate of MIT and received his Ph.D. From Cornell University.

Published by:

dreamtech PRESS

19-A, Ansari Road, Daryaganj, New Delhi-110002
Tel.: 91-11-23284212, 23243075
Fax: 91-11-23243078
Email: feedback@dreamtechpress.com;