

Fully updated for Java SE 8 (JDK 8)

Java

The Complete Reference

Ninth Edition



Comprehensive Coverage of the Java Language

Herbert Schildt

Oracle
Press™

ORACLE®

Oracle Press™

The
Complete
Reference

Java™
Ninth Edition

About the Author

Best-selling author **Herbert Schildt** has written extensively about programming for nearly three decades and is a leading authority on the Java language. His books have sold millions of copies worldwide and have been translated into all major foreign languages. He is the author of numerous books on Java, including *Java: A Beginner's Guide*, *Herb Schildt's Java Programming Cookbook*, and *Swing: A Beginner's Guide*. He has also written extensively about C, C++, and C#. Although interested in all facets of computing, his primary focus is computer languages, including compilers, interpreters, and robotic control languages. He also has an active interest in the standardization of languages. Schildt holds both graduate and undergraduate degrees from the University of Illinois. He can be reached at his consulting office at (217) 586-4683. His web site is www.HerbSchildt.com.

About the Technical Editor

Dr. Danny Coward has worked on all editions of the Java platform. He led the definition of Java Servlets into the first version of the Java EE platform and beyond, web services into the Java ME platform, and the strategy and planning for Java SE 7. He founded JavaFX technology and, most recently, designed the largest addition to the Java EE 7 standard, the Java WebSocket API. From coding in Java to designing APIs with industry experts, to serving for several years as an executive to the Java Community Process, he has a uniquely broad perspective into multiple aspects of Java technology. Additionally, he is the author of *JavaWebSocket Programming* and an upcoming book on Java EE. Dr. Coward holds bachelor's, master's, and doctorate's in mathematics from the University of Oxford.

ORACLE®

Oracle Press™

The
Complete
Reference

Java™
Ninth Edition

Herbert Schildt

**Mc
Graw
Hill
Education**

New York Chicago San Francisco
Athens London Madrid Mexico City
Milan New Delhi Singapore Sydney Toronto

Copyright © 2014 by McGraw-Hill Education (Publisher). All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of Publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

ISBN: 978-0-07-180856-9

MHID: 0-07-180856-6

e-Book conversion by Cenveo® Publisher Services

Version 1.0

The material in this eBook also appears in the print version of this title: ISBN: 978-0-071-80855-2,
MHID: 0-07-180855-8.

McGraw-Hill Education eBooks are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. To contact a representative, please visit the Contact Us pages at www.mhprofessional.com.

All trademarks are trademarks of their respective owners. Rather than put a trademark symbol after every occurrence of a trademarked name, we use names in an editorial fashion only, and to the benefit of the trademark owner, with no intention of infringement of the trademark. Where such designations appear in this book, they have been printed with initial caps.

Information has been obtained by McGraw-Hill Education from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, McGraw-Hill Education, or others, McGraw-Hill Education does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from the use of such information.

Oracle and Java are registered trademarks of Oracle Corporation and/or its affiliates. All other trademarks are the property of their respective owners, and McGraw-Hill Education makes no claim of ownership by the mention of products that contain these marks.

Screen displays of copyrighted Oracle software programs have been reproduced herein with the permission of Oracle Corporation and/or its affiliates.

TERMS OF USE

This is a copyrighted work and McGraw-Hill Education (“McGraw Hill”) and its licensors reserve all rights in and to the work. Use of this work is subject to these terms. Except as permitted under the Copyright Act of 1976 and the right to store and retrieve one copy of the work, you may not decompile, disassemble, reverse engineer, reproduce, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish or sublicense the work or any part of it without McGraw-Hill’s prior consent. You may use the work for your own noncommercial and personal use; any other use of the work is strictly prohibited. Your right to use the work may be terminated if you fail to comply with these terms.

THE WORK IS PROVIDED “AS IS.” McGRAW-HILL AND ITS LICENSORS MAKE NO GUARANTEES OR WARRANTIES AS TO THE ACCURACY, ADEQUACY OR COMPLETENESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. McGraw-Hill and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither McGraw-Hill nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. McGraw-Hill has no responsibility for the content of any information accessed through the work. Under no circumstances shall McGraw-Hill and/or its licensors be liable for any indirect, incidental, special, punitive, consequential or similar damages that result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.

Contents at a Glance

Part I

The Java Language

1	The History and Evolution of Java	3
2	An Overview of Java	17
3	Data Types, Variables, and Arrays	35
4	Operators	61
5	Control Statements	81
6	Introducing Classes	109
7	A Closer Look at Methods and Classes	129
8	Inheritance	161
9	Packages and Interfaces	187
10	Exception Handling	213
11	Multithreaded Programming	233
12	Enumerations, Autoboxing, and Annotations (Metadata)	263
13	I/O, Applets, and Other Topics	301
14	Generics	337
15	Lambda Expressions	381

Part II

The Java Library

16	String Handling	413
17	Exploring java.lang	441
18	java.util Part 1: The Collections Framework	497
19	java.util Part 2: More Utility Classes	579
20	Input/Output: Exploring java.io	641
21	Exploring NIO	689
22	Networking	727
23	The Applet Class	747
24	Event Handling	769
25	Introducing the AWT: Working with Windows, Graphics, and Text	797
26	Using AWT Controls, Layout Managers, and Menus	833
27	Images	885
28	The Concurrency Utilities	915
29	The Stream API	965
30	Regular Expressions and Other Packages	991

Part III	Introducing GUI Programming with Swing	
31	Introducing Swing	1021
32	Exploring Swing	1041
33	Introducing Swing Menus	1069
Part IV	Introducing GUI Programming with JavaFX	
34	Introducing JavaFX GUI Programming	1105
35	Exploring JavaFX Controls	1125
36	Introducing JavaFX Menus	1171
Part V	Applying Java	
37	Java Beans	1199
38	Introducing Servlets	1211
Appendix	Using Java's Documentation Comments	1235
	Index	1243

Contents

Preface	xxxii
Part I The Java Language	
Chapter 1 The History and Evolution of Java	3
Java's Lineage	3
The Birth of Modern Programming: C	4
C++: The Next Step	5
The Stage Is Set for Java	6
The Creation of Java	6
The C# Connection	8
How Java Changed the Internet	8
Java Applets	8
Security	9
Portability	9
Java's Magic: The Bytecode	9
Servlets: Java on the Server Side	10
The Java Buzzwords	10
Simple	11
Object-Oriented	11
Robust	11
Multithreaded	12
Architecture-Neutral	12
Interpreted and High Performance	12
Distributed	12
Dynamic	13
The Evolution of Java	13
Java SE 8	15
A Culture of Innovation	16
Chapter 2 An Overview of Java	17
Object-Oriented Programming	17
Two Paradigms	17
Abstraction	18
The Three OOP Principles	18

A First Simple Program	23
Entering the Program	23
Compiling the Program	23
A Closer Look at the First Sample Program.....	24
A Second Short Program.....	26
Two Control Statements.....	28
The if Statement	28
The for Loop	29
Using Blocks of Code.....	30
Lexical Issues	32
Whitespace.....	32
Identifiers.....	32
Literals	32
Comments	32
Separators.....	33
The Java Keywords.....	33
The Java Class Libraries.....	34
Chapter 3 Data Types, Variables, and Arrays	35
Java Is a Strongly Typed Language	35
The Primitive Types	35
Integers.....	36
byte	36
short	37
int	37
long.....	37
Floating-Point Types.....	38
float.....	38
double.....	38
Characters.....	39
Booleans	40
A Closer Look at Literals.....	41
Integer Literals.....	41
Floating-Point Literals.....	42
Boolean Literals.....	43
Character Literals	43
String Literals.....	43
Variables	44
Declaring a Variable	44
Dynamic Initialization	45
The Scope and Lifetime of Variables	45
Type Conversion and Casting	48
Java's Automatic Conversions	48
Casting Incompatible Types	48
Automatic Type Promotion in Expressions	50
The Type Promotion Rules	50

Arrays	51
One-Dimensional Arrays	51
Multidimensional Arrays	54
Alternative Array Declaration Syntax	58
A Few Words About Strings	58
A Note to C/C++ Programmers About Pointers	59
Chapter 4 Operators	61
Arithmetic Operators	61
The Basic Arithmetic Operators	62
The Modulus Operator	63
Arithmetic Compound Assignment Operators	63
Increment and Decrement	64
The Bitwise Operators	66
The Bitwise Logical Operators	67
The Left Shift	69
The Right Shift	70
The Unsigned Right Shift	72
Bitwise Operator Compound Assignments	73
Relational Operators	74
Boolean Logical Operators	75
Short-Circuit Logical Operators	76
The Assignment Operator	77
The ? Operator	77
Operator Precedence	78
Using Parentheses	79
Chapter 5 Control Statements	81
Java's Selection Statements	81
if	81
switch	84
Iteration Statements	89
while	89
do-while	90
for	93
The For-Each Version of the for Loop	97
Nested Loops	102
Jump Statements	102
Using break	102
Using continue	106
Chapter 6 Introducing Classes	109
Class Fundamentals	109
The General Form of a Class	109
A Simple Class	110
Declaring Objects	113
A Closer Look at new	113

Assigning Object Reference Variables	115
Introducing Methods.....	115
Adding a Method to the Box Class.....	116
Returning a Value	118
Adding a Method That Takes Parameters	119
Constructors.....	121
Parameterized Constructors.....	123
The this Keyword	124
Instance Variable Hiding.....	125
Garbage Collection	125
The finalize() Method	126
A Stack Class	126
Chapter 7 A Closer Look at Methods and Classes	129
Overloading Methods	129
Overloading Constructors.....	132
Using Objects as Parameters	134
A Closer Look at Argument Passing.....	136
Returning Objects	138
Recursion	139
Introducing Access Control.....	141
Understanding static	145
Introducing final	146
Arrays Revisited	147
Introducing Nested and Inner Classes	149
Exploring the String Class.....	152
Using Command-Line Arguments	154
Varargs: Variable-Length Arguments	155
Overloading Vararg Methods	158
Varargs and Ambiguity	159
Chapter 8 Inheritance	161
Inheritance Basics	161
Member Access and Inheritance	163
A More Practical Example.....	164
A Superclass Variable Can Reference a Subclass Object.....	166
Using super.....	167
Using super to Call Superclass Constructors	167
A Second Use for super.....	170
Creating a Multilevel Hierarchy	171
When Constructors Are Executed.....	174
Method Overriding	175
Dynamic Method Dispatch	178
Why Overridden Methods?	179
Applying Method Overriding	180

Using Abstract Classes	181
Using final with Inheritance	184
Using final to Prevent Overriding	184
Using final to Prevent Inheritance	185
The Object Class	185
Chapter 9 Packages and Interfaces	187
Packages	187
Defining a Package	188
Finding Packages and CLASSPATH	188
A Short Package Example	189
Access Protection	190
An Access Example	191
Importing Packages	194
Interfaces	196
Defining an Interface	196
Implementing Interfaces	197
Nested Interfaces	200
Applying Interfaces	201
Variables in Interfaces	204
Interfaces Can Be Extended	206
Default Interface Methods	207
Default Method Fundamentals	208
A More Practical Example	209
Multiple Inheritance Issues	210
Use static Methods in an Interface	211
Final Thoughts on Packages and Interfaces	212
Chapter 10 Exception Handling	213
Exception-Handling Fundamentals	213
Exception Types	214
Uncaught Exceptions	215
Using try and catch	216
Displaying a Description of an Exception	218
Multiple catch Clauses	218
Nested try Statements	220
throw	222
throws	223
finally	224
Java's Built-in Exceptions	226
Creating Your Own Exception Subclasses	227
Chained Exceptions	230
Three Recently Added Exception Features	231
Using Exceptions	232

Chapter 11 Multithreaded Programming	233
The Java Thread Model	234
Thread Priorities	235
Synchronization	235
Messaging	236
The Thread Class and the Runnable Interface	236
The Main Thread	237
Creating a Thread	238
Implementing Runnable	239
Extending Thread	241
Choosing an Approach	242
Creating Multiple Threads	242
Using isAlive() and join()	243
Thread Priorities	246
Synchronization	247
Using Synchronized Methods	247
The synchronized Statement	249
Interthread Communication	251
Deadlock	255
Suspending, Resuming, and Stopping Threads	257
Obtaining A Thread's State	259
Using Multithreading	261
Chapter 12 Enumerations, Autoboxing, and Annotations (Metadata)	263
Enumerations	263
Enumeration Fundamentals	263
The values() and valueOf() Methods	266
Java Enumerations Are Class Types	267
Enumerations Inherit Enum	269
Another Enumeration Example	271
Type Wrappers	272
Character	273
Boolean	273
The Numeric Type Wrappers	273
Autoboxing	274
Autoboxing and Methods	275
Autoboxing/Unboxing Occurs in Expressions	276
Autoboxing/Unboxing Boolean and Character Values	278
Autoboxing/Unboxing Helps Prevent Errors	278
A Word of Warning	279
Annotations (Metadata)	279
Annotation Basics	280
Specifying a Retention Policy	281
Obtaining Annotations at Run Time by Use of Reflection	281
The AnnotatedElement Interface	286
Using Default Values	287

Marker Annotations	288
Single-Member Annotations	289
The Built-In Annotations	290
Type Annotations	292
Repeating Annotations	297
Some Restrictions	299
Chapter 13 I/O, Applets, and Other Topics	301
I/O Basics	301
Streams	302
Byte Streams and Character Streams	302
The Predefined Streams	304
Reading Console Input	305
Reading Characters	305
Reading Strings	306
Writing Console Output	308
The PrintWriter Class	308
Reading and Writing Files	309
Automatically Closing a File	315
Applet Fundamentals	318
The transient and volatile Modifiers	322
Using instanceof	322
strictfp	324
Native Methods	325
Problems with Native Methods	328
Using assert	328
Assertion Enabling and Disabling Options	331
Static Import	331
Invoking Overloaded Constructors Through this()	334
Compact API Profiles	336
Chapter 14 Generics	337
What Are Generics?	338
A Simple Generics Example	338
Generics Work Only with Reference Types	342
Generic Types Differ Based on Their Type Arguments	342
How Generics Improve Type Safety	342
A Generic Class with Two Type Parameters	345
The General Form of a Generic Class	346
Bounded Types	346
Using Wildcard Arguments	349
Bounded Wildcards	352
Creating a Generic Method	356
Generic Constructors	359
Generic Interfaces	360
Raw Types and Legacy Code	362

Generic Class Hierarchies	364
Using a Generic Superclass	365
A Generic Subclass	367
Run-Time Type Comparisons Within a Generic Hierarchy	368
Casting	370
Overriding Methods in a Generic Class	371
Type Inference with Generics	372
Erasure	373
Bridge Methods	374
Ambiguity Errors	375
Some Generic Restrictions	377
Type Parameters Can't Be Instantiated	377
Restrictions on Static Members	377
Generic Array Restrictions	377
Generic Exception Restriction	379
Chapter 15 Lambda Expressions	381
Introducing Lambda Expressions	382
Lambda Expression Fundamentals	382
Functional Interfaces	383
Some Lambda Expression Examples	384
Block Lambda Expressions	387
Generic Functional Interfaces	389
Passing Lambda Expressions as Arguments	391
Lambda Expressions and Exceptions	394
Lambda Expressions and Variable Capture	395
Method References	396
Method References to static Methods	396
Method References to Instance Methods	397
Method References with Generics	401
Constructor References	404
Predefined Functional Interfaces	408
Part II The Java Library	
Chapter 16 String Handling	413
The String Constructors	414
String Length	416
Special String Operations	416
String Literals	416
String Concatenation	417
String Concatenation with Other Data Types	417
String Conversion and <code>toString()</code>	418
Character Extraction	419
<code>charAt()</code>	419
<code>getChars()</code>	419

getBytes()	420
toCharArray()	420
String Comparison	420
equals() and equalsIgnoreCase()	421
regionMatches()	421
startsWith() and endsWith()	422
equals() Versus ==	422
compareTo()	423
Searching Strings	424
Modifying a String	426
substring()	426
concat()	427
replace()	427
trim()	428
Data Conversion Using valueOf()	428
Changing the Case of Characters Within a String	429
Joining Strings	430
Additional String Methods	431
StringBuffer	432
StringBuffer Constructors	432
length() and capacity()	433
ensureCapacity()	433
setLength()	433
charAt() and setCharAt()	434
getChars()	434
append()	435
insert()	435
reverse()	436
delete() and deleteCharAt()	436
replace()	437
substring()	437
Additional StringBuffer Methods	438
StringBuilder	439
Chapter 17 Exploring java.lang	441
Primitive Type Wrappers	442
Number	442
Double and Float	442
Understanding isInfinite() and isNaN()	446
Byte, Short, Integer, and Long	447
Character	455
Additions to Character for Unicode Code Point Support	458
Boolean	458
Void	460
Process	460

Runtime	461
Memory Management	462
Executing Other Programs	464
ProcessBuilder	465
System	467
Using currentTimeMillis() to Time Program Execution	469
Using arraycopy()	469
Environment Properties	470
Object	471
Using clone() and the Cloneable Interface	471
Class	473
ClassLoader	477
Math	477
Trigonometric Functions	477
Exponential Functions	478
Rounding Functions	478
Miscellaneous Math Methods	479
StrictMath	481
Compiler	481
Thread, ThreadGroup, and Runnable	481
The Runnable Interface	481
Thread	482
ThreadGroup	484
ThreadLocal and InheritableThreadLocal	488
Package	489
RuntimePermission	490
Throwable	490
SecurityManager	490
StackTraceElement	491
Enum	492
ClassValue	493
The CharSequence Interface	493
The Comparable Interface	493
The Appendable Interface	494
The Iterable Interface	494
The Readable Interface	495
The AutoCloseable Interface	495
The Thread.UncaughtExceptionHandler Interface	495
The java.lang Subpackages	495
java.lang.annotation	496
java.lang.instrument	496
java.lang.invoke	496
java.lang.management	496
java.lang.ref	496
java.lang.reflect	496

Chapter 18 java.util Part 1: The Collections Framework	497
Collections Overview	498
JDK 5 Changed the Collections Framework.	500
Generics Fundamentally Changed the Collections Framework	500
Autoboxing Facilitates the Use of Primitive Types	500
The For-Each Style for Loop	500
The Collection Interfaces	501
The Collection Interface	501
The List Interface	504
The Set Interface	504
The SortedSet Interface	506
The NavigableSet Interface	507
The Queue Interface	508
The Deque Interface	509
The Collection Classes	510
The ArrayList Class	511
The LinkedList Class	515
The HashSet Class	516
The LinkedHashSet Class	517
The TreeSet Class	518
The PriorityQueue Class	519
The ArrayDeque Class	520
The EnumSet Class	521
Accessing a Collection via an Iterator	521
Using an Iterator	523
The For-Each Alternative to Iterators	525
Spliterators	526
Storing User-Defined Classes in Collections	529
The RandomAccess Interface	530
Working with Maps	530
The Map Interfaces	531
The Map Classes	537
Comparators	542
Using a Comparator	544
The Collection Algorithms	550
Arrays	556
The Legacy Classes and Interfaces	561
The Enumeration Interface	562
Vector	562
Stack	566
Dictionary	568
Hashtable	569
Properties	572
Using store() and load()	576
Parting Thoughts on Collections	577

Chapter 19 java.util Part 2: More Utility Classes	579
StringTokenizer	579
BitSet	581
Optional, OptionalDouble, OptionalInt, and OptionalLong	584
Date	586
Calendar	588
GregorianCalendar	591
TimeZone	593
SimpleTimeZone	594
Locale	594
Random	596
Observable	598
The Observer Interface	599
An Observer Example	599
Timer and TimerTask	602
Currency	604
Formatter	605
The Formatter Constructors	605
The Formatter Methods	606
Formatting Basics	607
Formatting Strings and Characters	609
Formatting Numbers	609
Formatting Time and Date	610
The %n and %% Specifiers	612
Specifying a Minimum Field Width	612
Specifying Precision	614
Using the Format Flags	614
Justifying Output	615
The Space, +, 0, and (Flags	616
The Comma Flag	617
The # Flag	617
The Uppercase Option	617
Using an Argument Index	618
Closing a Formatter	619
The Java printf() Connection	620
Scanner	620
The Scanner Constructors	620
Scanning Basics	620
Some Scanner Examples	624
Setting Delimiters	628
Other Scanner Features	629
The ResourceBundle, ListResourceBundle, and PropertyResourceBundle Classes	630
Miscellaneous Utility Classes and Interfaces	635

The java.util Subpackages	635
java.util.concurrent, java.util.concurrent.atomic, and java.util.concurrent.locks	636
java.util.function	636
java.util.jar	639
java.util.logging	639
java.util.prefs	639
java.util.regex	639
java.util.spi	639
java.util.stream	639
java.util.zip	639
Chapter 20 Input/Output: Exploring java.io	641
The I/O Classes and Interfaces	641
File	642
Directories	645
Using FilenameFilter	646
The listFiles() Alternative	647
Creating Directories	648
The AutoCloseable, Closeable, and Flushable Interfaces	648
I/O Exceptions	649
Two Ways to Close a Stream	649
The Stream Classes	650
The Byte Streams	651
InputStream	651
OutputStream	651
FileInputStream	652
FileOutputStream	654
ByteArrayInputStream	656
ByteArrayOutputStream	658
Filtered Byte Streams	659
Buffered Byte Streams	659
SequenceInputStream	663
PrintStream	665
DataOutputStream and DataInputStream	667
RandomAccessFile	669
The Character Streams	670
Reader	670
Writer	670
FileReader	672
FileWriter	673
CharArrayReader	674
CharArrayWriter	675
BufferedReader	676
BufferedWriter	678

PushbackReader	678
PrintWriter	679
The Console Class	680
Serialization	682
Serializable	682
Externalizable	683
ObjectOutput	683
ObjectOutputStream	684
ObjectInput	685
ObjectInputStream	685
A Serialization Example	686
Stream Benefits	688
Chapter 21 Exploring NIO	689
The NIO Classes	689
NIO Fundamentals	690
Buffers	690
Channels	691
Charsets and Selectors	693
Enhancements Added to NIO by JDK 7	694
The Path Interface	694
The Files Class	695
The Paths Class	698
The File Attribute Interfaces	698
The FileSystem, FileSystems, and FileStore Classes	700
Using the NIO System	700
Use NIO for Channel-Based I/O	700
Use NIO for Stream-Based I/O	709
Use NIO for Path and File System Operations	712
Pre-JDK 7 Channel-Based Examples	719
Read a File, Pre-JDK 7	720
Write to a File, Pre-JDK 7	723
Chapter 22 Networking	727
Networking Basics	727
The Networking Classes and Interfaces	728
InetAddress	729
Factory Methods	729
Instance Methods	730
Inet4Address and Inet6Address	731
TCP/IP Client Sockets	731
URL	735
URLConnection	736
HttpURLConnection	739
The URI Class	741
Cookies	741

TCP/IP Server Sockets.....	741
Datagrams.....	742
DatagramSocket.....	742
DatagramPacket.....	743
A Datagram Example.....	744
Chapter 23 The Applet Class	747
Two Types of Applets	747
Applet Basics.....	747
The Applet Class	749
Applet Architecture	751
An Applet Skeleton	751
Applet Initialization and Termination	753
Overriding update().....	754
Simple Applet Display Methods	754
Requesting Repainting.....	756
A Simple Banner Applet.....	757
Using the Status Window.....	759
The HTML APPLET Tag.....	760
Passing Parameters to Applets.....	761
Improving the Banner Applet.....	763
getDocumentBase() and getCodeBase()	764
AppletContext and showDocument()	765
The AudioClip Interface	767
The AppletStub Interface	767
Outputting to the Console	767
Chapter 24 Event Handling	769
Two Event Handling Mechanisms.....	769
The Delegation Event Model	770
Events	770
Event Sources.....	770
Event Listeners.....	771
Event Classes.....	771
The ActionEvent Class.....	773
The AdjustmentEvent Class	773
The ComponentEvent Class	774
The ContainerEvent Class.....	774
The FocusEvent Class	775
The InputEvent Class	775
The ItemEvent Class	776
The KeyEvent Class	777
The MouseEvent Class.....	778
The MouseWheelEvent Class	779
The TextEvent Class.....	780
The WindowEvent Class	780

Sources of Events	781
Event Listener Interfaces.....	782
The ActionListener Interface	783
The AdjustmentListener Interface.....	783
The ComponentListener Interface	783
The ContainerListener Interface	783
The FocusListener Interface.....	783
The ItemListener Interface.....	783
The KeyListener Interface	784
The MouseListener Interface	784
The MouseMotionListener Interface.....	784
The MouseWheelListener Interface.....	784
The TextListener Interface	784
The WindowFocusListener Interface.....	785
The WindowListener Interface.....	785
Using the Delegation Event Model.....	785
Handling Mouse Events	785
Handling Keyboard Events	788
Adapter Classes.....	791
Inner Classes.....	793
Anonymous Inner Classes.....	795
Chapter 25 Introducing the AWT: Working with Windows, Graphics, and Text	797
AWT Classes	798
Window Fundamentals	800
Component	800
Container	801
Panel	801
Window	801
Frame	801
Canvas	801
Working with Frame Windows.....	802
Setting the Window's Dimensions	802
Hiding and Showing a Window	802
Setting a Window's Title	802
Closing a Frame Window.....	803
Creating a Frame Window in an AWT-Based Applet	803
Handling Events in a Frame Window.....	805
Creating a Windowed Program	809
Displaying Information Within a Window	811
Introducing Graphics	811
Drawing Lines	811
Drawing Rectangles.....	812
Drawing Ellipses and Circles.....	812
Drawing Arcs	812

Drawing Polygons	813
Demonstrating the Drawing Methods	813
Sizing Graphics	814
Working with Color	815
Color Methods.....	816
Setting the Current Graphics Color.....	817
A Color Demonstration Applet.....	817
Setting the Paint Mode	818
Working with Fonts	819
Determining the Available Fonts	821
Creating and Selecting a Font.....	822
Obtaining Font Information.....	824
Managing Text Output Using FontMetrics.....	825
Displaying Multiple Lines of Text.....	825
Centering Text.....	828
Multiline Text Alignment	829
Chapter 26 Using AWT Controls, Layout Managers, and Menus.....	833
AWT Control Fundamentals	834
Adding and Removing Controls.....	834
Responding to Controls	834
The HeadlessException.....	835
Labels	835
Using Buttons.....	836
Handling Buttons	836
Applying Check Boxes.....	840
Handling Check Boxes	840
CheckboxGroup.....	842
Choice Controls	844
Handling Choice Lists.....	844
Using Lists.....	846
Handling Lists	847
Managing Scroll Bars.....	849
Handling Scroll Bars	850
Using a TextField	852
Handling a TextField.....	853
Using a TextArea	854
Understanding Layout Managers	855
FlowLayout.....	856
BorderLayout.....	858
Using Insets	860
GridLayout.....	861
CardLayout.....	862
GridBagLayout.....	865
Menu Bars and Menus.....	870

Dialog Boxes.....	876
FileDialog	880
A Word About Overriding <code>paint()</code>	882
Chapter 27 Images	885
File Formats	885
Image Fundamentals: Creating, Loading, and Displaying	886
Creating an Image Object.....	886
Loading an Image	886
Displaying an Image	887
ImageObserver.....	888
Double Buffering	889
MediaTracker	892
ImageProducer.....	895
MemoryImageSource	895
ImageConsumer.....	897
PixelGrabber	897
ImageFilter	899
CropImageFilter	900
RGBImageFilter.....	902
Additional Imaging Classes	913
Chapter 28 The Concurrency Utilities	915
The Concurrent API Packages	916
<code>java.util.concurrent</code>	916
<code>java.util.concurrent.atomic</code>	917
<code>java.util.concurrent.locks</code>	917
Using Synchronization Objects.....	917
Semaphore.....	918
CountDownLatch.....	923
CyclicBarrier.....	925
Exchanger	927
Phaser	930
Using an Executor	937
A Simple Executor Example	937
Using Callable and Future	939
The TimeUnit Enumeration	942
The Concurrent Collections	943
Locks.....	943
Atomic Operations.....	946
Parallel Programming via the Fork/Join Framework	947
The Main Fork/Join Classes	948
The Divide-and-Conquer Strategy	951
A Simple First Fork/Join Example	952
Understanding the Impact of the Level of Parallelism	955
An Example that Uses <code>RecursiveTask<V></code>	958

Executing a Task Asynchronously.....	960
Cancelling a Task.....	961
Determining a Task's Completion Status.....	961
Restarting a Task	961
Things to Explore	962
Some Fork/Join Tips.....	963
The Concurrency Utilities Versus Java's Traditional Approach	964
Chapter 29 The Stream API	965
Stream Basics	965
Stream Interfaces.....	966
How to Obtain a Stream	969
A Simple Stream Example.....	969
Reduction Operations	973
Using Parallel Streams.....	975
Mapping	978
Collecting	982
Iterators and Streams.....	986
Use an Iterator with a Stream.....	986
Use Spliterator.....	987
More to Explore in the Stream API	990
Chapter 30 Regular Expressions and Other Packages	991
The Core Java API Packages	991
Regular Expression Processing	993
Pattern	994
Matcher	994
Regular Expression Syntax	995
Demonstrating Pattern Matching.....	995
Two Pattern-Matching Options.....	1001
Exploring Regular Expressions.....	1001
Reflection	1001
Remote Method Invocation (RMI).....	1005
A Simple Client/Server Application Using RMI	1006
Formatting Date and Time with java.text.....	1009
DateFormat Class.....	1009
SimpleDateFormat Class	1011
The Time and Date API Added by JDK 8.....	1013
Time and Date Fundamentals.....	1013
Formatting Date and Time	1015
Parsing Date and Time Strings	1017
Other Things to Explore in java.time.....	1018

Part III Introducing GUI Programming with Swing

Chapter 31	Introducing Swing	1021
	The Origins of Swing	1021
	Swing Is Built on the AWT	1022
	Two Key Swing Features	1022
	Swing Components Are Lightweight	1022
	Swing Supports a Pluggable Look and Feel	1022
	The MVC Connection	1023
	Components and Containers	1024
	Components	1024
	Containers	1025
	The Top-Level Container Panes	1025
	The Swing Packages	1026
	A Simple Swing Application	1026
	Event Handling	1030
	Create a Swing Applet	1033
	Painting in Swing	1036
	Painting Fundamentals	1036
	Compute the Paintable Area	1037
	A Paint Example	1037
Chapter 32	Exploring Swing	1041
	JLabel and ImageIcon	1041
	JTextField	1043
	The Swing Buttons	1045
	JButton	1045
	JToggleButton	1047
	Check Boxes	1049
	Radio Buttons	1051
	JTabbedPane	1053
	JScrollPane	1056
	JList	1058
	JComboBox	1061
	Trees	1063
	JTable	1066
Chapter 33	Introducing Swing Menus	1069
	Menu Basics	1069
	An Overview of JMenuBar, JMenu, and JMenuItem	1071
	JMenuBar	1071
	JMenu	1072
	JMenuItem	1073
	Create a Main Menu	1074
	Add Mnemonics and Accelerators to Menu Items	1078
	Add Images and Tooltips to Menu Items	1080
	Use JRadioButtonMenuItem and JCheckBoxMenuItem	1081
	Create a Popup Menu	1083

Create a Toolbar	1087
Use Actions	1089
Put the Entire MenuDemo Program Together	1095
Continuing Your Exploration of Swing	1101
 Part IV Introducing GUI Programming with JavaFX	
Chapter 34 Introducing JavaFX GUI Programming	1105
JavaFX Basic Concepts	1106
The JavaFX Packages	1106
The Stage and Scene Classes	1106
Nodes and Scene Graphs	1107
Layouts	1107
The Application Class and the Lifecycle Methods	1107
Launching a JavaFX Application	1108
A JavaFX Application Skeleton	1108
Compiling and Running a JavaFX Program	1111
The Application Thread	1112
A Simple JavaFX Control: Label	1112
Using Buttons and Events	1114
Event Basics	1115
Introducing the Button Control	1115
Demonstrating Event Handling and the Button	1116
Drawing Directly on a Canvas	1119
Chapter 35 Exploring JavaFX Controls	1125
Using Image and ImageView	1125
Adding an Image to a Label	1128
Using an Image with a Button	1130
ToggleButton	1133
RadioButton	1135
Handling Change Events in a Toggle Group	1138
An Alternative Way to Handle Radio Buttons	1139
CheckBox	1142
ListView	1146
ListView Scrollbars	1149
Enabling Multiple Selections	1150
ComboBox	1151
TextField	1154
ScrollPane	1157
TreeView	1160
Introducing Effects and Transforms	1164
Effects	1165
Transforms	1166
Demonstrating Effects and Transforms	1167
Adding Tooltips	1170
Disabling a Control	1170

Chapter 36	Introducing JavaFX Menus.	1171
	Menu Basics	1171
	An Overview ofMenuBar, Menu, and MenuItem	1173
	MenuBar.	1173
	Menu.	1174
	MenuItem.	1174
	Create a Main Menu	1175
	Add Mnemonics and Accelerators to Menu Items.	1180
	Add Images to Menu Items	1182
	Use RadioMenuItem and CheckMenuItem	1183
	Create a Context Menu	1185
	Create a Toolbar.	1189
	Put the Entire MenuDemo Program Together	1191
	Continuing Your Exploration of JavaFX.	1196

Part V Applying Java

Chapter 37	Java Beans	1199
	What Is a Java Bean?.	1199
	Advantages of Java Beans.	1200
	Introspection	1200
	Design Patterns for Properties	1200
	Design Patterns for Events	1202
	Methods and Design Patterns.	1202
	Using the BeanInfo Interface.	1202
	Bound and Constrained Properties	1203
	Persistence	1203
	Customizers	1203
	The Java Beans API	1204
	Introspector	1206
	PropertyDescriptor	1206
	EventSetDescriptor	1206
	MethodDescriptor.	1206
	A Bean Example.	1206
Chapter 38	Introducing Servlets.	1211
	Background	1211
	The Life Cycle of a Servlet.	1212
	Servlet Development Options.	1212
	Using Tomcat	1213
	A Simple Servlet.	1214
	Create and Compile the Servlet Source Code.	1215
	Start Tomcat.	1215
	Start a Web Browser and Request the Servlet	1216

The Servlet API	1216
The javax.servlet Package	1216
The Servlet Interface	1217
The ServletConfig Interface	1218
The ServletContext Interface	1218
The ServletRequest Interface	1218
The ServletResponse Interface	1218
The GenericServlet Class	1220
The ServletInputStream Class	1220
The ServletOutputStream Class	1220
The ServletException Classes	1220
Reading Servlet Parameters	1220
The javax.servlet.http Package	1222
The HttpServletRequest Interface	1222
The HttpServletResponse Interface	1222
The HttpSession Interface	1223
The Cookie Class	1224
The HttpServlet Class	1225
Handling HTTP Requests and Responses	1227
Handling HTTP GET Requests	1227
Handling HTTP POST Requests	1229
Using Cookies	1230
Session Tracking	1232
Appendix	
Using Java's Documentation Comments	1235
The javadoc Tags	1235
@author	1236
{@code}	1236
@deprecated	1236
{@docRoot}	1237
@exception	1237
{@inheritDoc}	1237
{@link}	1237
{@linkplain}	1237
{@literal}	1237
@param	1237
@return	1238
@see	1238
@serial	1238
@serialData	1238
@serialField	1238
@since	1238

@throws	1239
{@value}	1239
@version	1239
The General Form of a Documentation Comment.....	1239
What javadoc Outputs	1239
An Example that Uses Documentation Comments.....	1240
Index	1243

Preface

Java is one of the world's most important and widely used computer languages. Furthermore, it has held that distinction for many years. Unlike some other computer languages whose influence has waned with the passage of time, Java's has grown stronger. Java leapt to the forefront of Internet programming with its first release. Each subsequent version has solidified that position. Today, it is still the first and best choice for developing web-based applications. Simply put: much of the modern world runs on Java code. Java really is that important.

A key reason for Java's success is its agility. Since its original 1.0 release, Java has continually adapted to changes in the programming environment and to changes in the way that programmers program. Most importantly, it has not just followed the trends, *it has helped create them*. Java's ability to accommodate the fast rate of change in the computing world is a crucial part of why it has been and continues to be so successful.

Since this book was first published in 1996, it has gone through several editions, each reflecting the ongoing evolution of Java. This is the Ninth edition, and it has been updated for Java SE 8 (JDK 8). As a result, this edition of the book contains a substantial amount of new material because Java SE 8 adds several new features to the Java language. The most important is the lambda expression, which introduces an entirely new syntax element and fundamentally increases the expressive power of the language. Because the impact of lambda expressions is so significant, an entire chapter is devoted to them. Furthermore, examples of their use are found elsewhere in the book. The lambda expression was also the catalyst for other new features. One is the stream library in **java.util.stream**, which supports pipeline operations on data. It too has an entire chapter devoted to it. Another is the default method, which makes it possible to add default functionality to an interface. Features such as repeating and type annotations further expand the power of Java. Java SE 8 also makes significant enhancements to the Java API library, several of which are described in this book.

Another important addition to this edition of the book is coverage of JavaFX, Java's new GUI framework. Because of the significant role that JavaFX is expected to play in the way Java applications are designed, three new chapters are devoted to it. Simply put, experience with JavaFX is something that Java programmers need. An additional chapter about Swing has also been included that discusses menus. Although Swing may ultimately be replaced by JavaFX, it is (at the time of this writing) still the most widely used Java GUI framework. Thus, expanded coverage was warranted. Finally, many small updates have been made throughout the book.

A Book for All Programmers

This book is for all programmers, whether you are a novice or an experienced pro. The beginner will find its carefully paced discussions and many examples especially helpful. Its in-depth coverage of Java's more advanced features and libraries will appeal to the pro. For both, it offers a lasting resource and handy reference.

What's Inside

This book is a comprehensive guide to the Java language, describing its syntax, keywords, and fundamental programming principles. Significant portions of the Java API library are also examined. The book is divided into five parts, each focusing on a different aspect of the Java programming environment.

Part I presents an in-depth tutorial of the Java language. It begins with the basics, including such things as data types, operators, control statements, and classes. It then moves on to inheritance, packages, interfaces, exception handling, and multithreading. Next, it describes annotations, enumerations, autoboxing, and generics. I/O and applets are also introduced. The final chapter in Part I covers lambda expressions. As mentioned, the lambda expression is the single most important new feature in Java SE 8.

Part II examines key aspects of Java's standard API library. Topics include strings, I/O, networking, the standard utilities, the Collections Framework, applets, the AWT, event handling, imaging, concurrency (including the Fork/Join Framework), regular expressions, and the new stream library.

Part III offers three chapters that introduce Swing.

Part IV presents three chapters that introduce JavaFX.

Part V contains two chapters that show examples of Java in action. The first discusses Java Beans. The second presents an introduction to servlets.

Don't Forget: Code on the Web

Remember, the source code for all of the examples in this book is available free-of-charge on the Web at www.oraclepressbooks.com.

Special Thanks

I want to give special thanks to Patrick Naughton, Joe O'Neil, and Danny Coward.

Patrick Naughton was one of the creators of the Java language. He also helped write the first edition of this book. For example, among many other contributions, much of the material in Chapters 20, 22, and 27 was initially provided by Patrick. His insights, expertise, and energy contributed greatly to the success of that book.

During the preparation of the second and third editions of this book, Joe O'Neil provided initial drafts for the material now found in Chapters 30, 32, 37, and 38 of this edition. Joe helped on several of my books and his input has always been top-notch.

Danny Coward is the technical editor for this edition of the book. Danny has worked on several of my books and his advice, insights, and suggestions have always been of great value and much appreciated.

HERBERT SCHILDT

For Further Study

Java: The Complete Reference is your gateway to the Herb Schildt series of Java programming books. Here are others that you will find of interest:

Herb Schildt's Java Programming Cookbook

Java: A Beginner's Guide

Swing: A Beginner's Guide

The Art of Java

PART

I

The Java Language

CHAPTER 1

The History and Evolution
of Java

CHAPTER 2

An Overview of Java

CHAPTER 3

Data Types, Variables,
and Arrays

CHAPTER 4

Operators

CHAPTER 5

Control Statements

CHAPTER 6

Introducing Classes

CHAPTER 7

A Closer Look at Methods
and Classes

CHAPTER 8

Inheritance

CHAPTER 9

Packages and Interfaces

CHAPTER 10

Exception Handling

CHAPTER 11

Multithreaded Programming

CHAPTER 12

Enumerations, Autoboxing,
and Annotations (Metadata)

CHAPTER 13

I/O, Applets, and
Other Topics

CHAPTER 14

Generics

CHAPTER 15

Lambda Expressions

CHAPTER

1

The History and Evolution of Java

To fully understand Java, one must understand the reasons behind its creation, the forces that shaped it, and the legacy that it inherits. Like the successful computer languages that came before, Java is a blend of the best elements of its rich heritage combined with the innovative concepts required by its unique mission. While the remaining chapters of this book describe the practical aspects of Java—including its syntax, key libraries, and applications—this chapter explains how and why Java came about, what makes it so important, and how it has evolved over the years.

Although Java has become inseparably linked with the online environment of the Internet, it is important to remember that Java is first and foremost a programming language. Computer language innovation and development occurs for two fundamental reasons:

- To adapt to changing environments and uses
- To implement refinements and improvements in the art of programming

As you will see, the development of Java was driven by both elements in nearly equal measure.

Java's Lineage

Java is related to C++, which is a direct descendant of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java's object-oriented features were influenced by C++. In fact, several of Java's defining characteristics come from—or are responses to—its predecessors. Moreover, the creation of Java was deeply rooted in the process of refinement and adaptation that has been occurring in computer programming languages for the past several decades. For these reasons, this section reviews the sequence of events and forces that led to Java. As you will see, each innovation in language design was driven by the need to solve a fundamental problem that the preceding languages could not solve. Java is no exception.

The Birth of Modern Programming: C

The C language shook the computer world. Its impact should not be underestimated, because it fundamentally changed the way programming was approached and thought about. The creation of C was a direct result of the need for a structured, efficient, high-level language that could replace assembly code when creating systems programs. As you probably know, when a computer language is designed, trade-offs are often made, such as the following:

- Ease-of-use versus power
- Safety versus efficiency
- Rigidity versus extensibility

Prior to C, programmers usually had to choose between languages that optimized one set of traits or the other. For example, although FORTRAN could be used to write fairly efficient programs for scientific applications, it was not very good for system code. And while BASIC was easy to learn, it wasn't very powerful, and its lack of structure made its usefulness questionable for large programs. Assembly language can be used to produce highly efficient programs, but it is not easy to learn or use effectively. Further, debugging assembly code can be quite difficult.

Another compounding problem was that early computer languages such as BASIC, COBOL, and FORTRAN were not designed around structured principles. Instead, they relied upon the GOTO as a primary means of program control. As a result, programs written using these languages tended to produce “spaghetti code”—a mass of tangled jumps and conditional branches that make a program virtually impossible to understand. While languages like Pascal are structured, they were not designed for efficiency, and failed to include certain features necessary to make them applicable to a wide range of programs. (Specifically, given the standard dialects of Pascal available at the time, it was not practical to consider using Pascal for systems-level code.)

So, just prior to the invention of C, no one language had reconciled the conflicting attributes that had dogged earlier efforts. Yet the need for such a language was pressing. By the early 1970s, the computer revolution was beginning to take hold, and the demand for software was rapidly outpacing programmers' ability to produce it. A great deal of effort was being expended in academic circles in an attempt to create a better computer language. But, and perhaps most importantly, a secondary force was beginning to be felt. Computer hardware was finally becoming common enough that a critical mass was being reached. No longer were computers kept behind locked doors. For the first time, programmers were gaining virtually unlimited access to their machines. This allowed the freedom to experiment. It also allowed programmers to begin to create their own tools. On the eve of C's creation, the stage was set for a quantum leap forward in computer languages.

Invented and first implemented by Dennis Ritchie on a DEC PDP-11 running the UNIX operating system, C was the result of a development process that started with an older language called BCPL, developed by Martin Richards. BCPL influenced a language called B, invented by Ken Thompson, which led to the development of C in the 1970s. For many years, the de facto standard for C was the one supplied with the UNIX operating system and described in *The C Programming Language* by Brian Kernighan and Dennis Ritchie (Prentice-Hall, 1978). C was formally standardized in December 1989, when the American National Standards Institute (ANSI) standard for C was adopted.

The creation of C is considered by many to have marked the beginning of the modern age of computer languages. It successfully synthesized the conflicting attributes that had so troubled earlier languages. The result was a powerful, efficient, structured language that was relatively easy to learn. It also included one other, nearly intangible aspect: it was a *programmer's language*. Prior to the invention of C, computer languages were generally designed either as academic exercises or by bureaucratic committees. C is different. It was designed, implemented, and developed by real, working programmers, reflecting the way that they approached the job of programming. Its features were honed, tested, thought about, and rethought by the people who actually used the language. The result was a language that programmers liked to use. Indeed, C quickly attracted many followers who had a near-religious zeal for it. As such, it found wide and rapid acceptance in the programmer community. In short, C is a language designed by and for programmers. As you will see, Java inherited this legacy.

C++: The Next Step

During the late 1970s and early 1980s, C became the dominant computer programming language, and it is still widely used today. Since C is a successful and useful language, you might ask why a need for something else existed. The answer is *complexity*. Throughout the history of programming, the increasing complexity of programs has driven the need for better ways to manage that complexity. C++ is a response to that need. To better understand why managing program complexity is fundamental to the creation of C++, consider the following.

Approaches to programming have changed dramatically since the invention of the computer. For example, when computers were first invented, programming was done by manually toggling in the binary machine instructions by use of the front panel. As long as programs were just a few hundred instructions long, this approach worked. As programs grew, assembly language was invented so that a programmer could deal with larger, increasingly complex programs by using symbolic representations of the machine instructions. As programs continued to grow, high-level languages were introduced that gave the programmer more tools with which to handle complexity.

The first widespread language was, of course, FORTRAN. While FORTRAN was an impressive first step, it is hardly a language that encourages clear and easy-to-understand programs. The 1960s gave birth to *structured programming*. This is the method of programming championed by languages such as C. The use of structured languages enabled programmers to write, for the first time, moderately complex programs fairly easily. However, even with structured programming methods, once a project reaches a certain size, its complexity exceeds what a programmer can manage. By the early 1980s, many projects were pushing the structured approach past its limits. To solve this problem, a new way to program was invented, called *object-oriented programming (OOP)*. Object-oriented programming is discussed in detail later in this book, but here is a brief definition: OOP is a programming methodology that helps organize complex programs through the use of inheritance, encapsulation, and polymorphism.

In the final analysis, although C is one of the world's great programming languages, there is a limit to its ability to handle complexity. Once the size of a program exceeds a certain point, it becomes so complex that it is difficult to grasp as a totality. While the precise size at which this occurs differs, depending upon both the nature of the program and the programmer, there is always a threshold at which a program becomes unmanageable.

C++ added features that enabled this threshold to be broken, allowing programmers to comprehend and manage larger programs.

C++ was invented by Bjarne Stroustrup in 1979, while he was working at Bell Laboratories in Murray Hill, New Jersey. Stroustrup initially called the new language “C with Classes.” However, in 1983, the name was changed to C++. C++ extends C by adding object-oriented features. Because C++ is built on the foundation of C, it includes all of C’s features, attributes, and benefits. This is a crucial reason for the success of C++ as a language. The invention of C++ was not an attempt to create a completely new programming language. Instead, it was an enhancement to an already highly successful one.

The Stage Is Set for Java

By the end of the 1980s and the early 1990s, object-oriented programming using C++ took hold. Indeed, for a brief moment it seemed as if programmers had finally found the perfect language. Because C++ blended the high efficiency and stylistic elements of C with the object-oriented paradigm, it was a language that could be used to create a wide range of programs. However, just as in the past, forces were brewing that would, once again, drive computer language evolution forward. Within a few years, the World Wide Web and the Internet would reach critical mass. This event would precipitate another revolution in programming.

The Creation of Java

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called “Oak,” but was renamed “Java” in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.

Somewhat surprisingly, the original impetus for Java was not the Internet! Instead, the primary motivation was the need for a platform-independent (that is, architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls. As you can probably guess, many different types of CPUs are used as controllers. The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time-consuming to create. An easier—and more cost-efficient—solution was needed. In an attempt to find such a solution, Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the creation of Java.

About the time that the details of Java were being worked out, a second, and ultimately more important, factor was emerging that would play a crucial role in the future of Java. This second force was, of course, the World Wide Web. Had the Web not taken shape at about the same time that Java was being implemented, Java might have remained a useful but obscure language for programming consumer electronics. However, with the emergence

of the World Wide Web, Java was propelled to the forefront of computer language design, because the Web, too, demanded portable programs.

Most programmers learn early in their careers that portable programs are as elusive as they are desirable. While the quest for a way to create efficient, portable (platform-independent) programs is nearly as old as the discipline of programming itself, it had taken a back seat to other, more pressing problems. Further, because (at that time) much of the computer world had divided itself into the three competing camps of Intel, Macintosh, and UNIX, most programmers stayed within their fortified boundaries, and the urgent need for portable code was reduced. However, with the advent of the Internet and the Web, the old problem of portability returned with a vengeance. After all, the Internet consists of a diverse, distributed universe populated with various types of computers, operating systems, and CPUs. Even though many kinds of platforms are attached to the Internet, users would like them all to be able to run the same program. What was once an irritating but low-priority problem had become a high-profile necessity.

By 1993, it became obvious to members of the Java design team that the problems of portability frequently encountered when creating code for embedded controllers are also found when attempting to create code for the Internet. In fact, the same problem that Java was initially designed to solve on a small scale could also be applied to the Internet on a large scale. This realization caused the focus of Java to switch from consumer electronics to Internet programming. So, while the desire for an architecture-neutral programming language provided the initial spark, the Internet ultimately led to Java's large-scale success.

As mentioned earlier, Java derives much of its character from C and C++. This is by intent. The Java designers knew that using the familiar syntax of C and echoing the object-oriented features of C++ would make their language appealing to the legions of experienced C/C++ programmers. In addition to the surface similarities, Java shares some of the other attributes that helped make C and C++ successful. First, Java was designed, tested, and refined by real, working programmers. It is a language grounded in the needs and experiences of the people who devised it. Thus, Java is a programmer's language. Second, Java is cohesive and logically consistent. Third, except for those constraints imposed by the Internet environment, Java gives you, the programmer, full control. If you program well, your programs reflect it. If you program poorly, your programs reflect that, too. Put differently, Java is not a language with training wheels. It is a language for professional programmers.

Because of the similarities between Java and C++, it is tempting to think of Java as simply the "Internet version of C++." However, to do so would be a large mistake. Java has significant practical and philosophical differences. While it is true that Java was influenced by C++, it is not an enhanced version of C++. For example, Java is neither upwardly nor downwardly compatible with C++. Of course, the similarities with C++ are significant, and if you are a C++ programmer, then you will feel right at home with Java. One other point: Java was not designed to replace C++. Java was designed to solve a certain set of problems. C++ was designed to solve a different set of problems. Both will coexist for many years to come.

As mentioned at the start of this chapter, computer languages evolve for two reasons: to adapt to changes in environment and to implement advances in the art of programming. The environmental change that prompted Java was the need for platform-independent programs destined for distribution on the Internet. However, Java also embodies changes in the way that people approach the writing of programs. For example, Java enhanced and refined the object-oriented paradigm used by C++, added integrated support for multithreading, and provided a library that simplified Internet access. In the final analysis,

though, it was not the individual features of Java that made it so remarkable. Rather, it was the language as a whole. Java was the perfect response to the demands of the then newly emerging, highly distributed computing universe. Java was to Internet programming what C was to system programming: a revolutionary force that changed the world.

The C# Connection

The reach and power of Java continues to be felt in the world of computer language development. Many of its innovative features, constructs, and concepts have become part of the baseline for any new language. The success of Java is simply too important to ignore.

Perhaps the most important example of Java's influence is C#. Created by Microsoft to support the .NET Framework, C# is closely related to Java. For example, both share the same general syntax, support distributed programming, and utilize the same object model. There are, of course, differences between Java and C#, but the overall "look and feel" of these languages is very similar. This "cross-pollination" from Java to C# is the strongest testimonial to date that Java redefined the way we think about and use a computer language.

How Java Changed the Internet

The Internet helped catapult Java to the forefront of programming, and Java, in turn, had a profound effect on the Internet. In addition to simplifying web programming in general, Java innovated a new type of networked program called the applet that changed the way the online world thought about content. Java also addressed some of the thorniest issues associated with the Internet: portability and security. Let's look more closely at each of these.

Java Applets

An *applet* is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser. Furthermore, an applet is downloaded on demand, without further interaction with the user. If the user clicks a link that contains an applet, the applet will be automatically downloaded and run in the browser. Applets are intended to be small programs. They are typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that execute locally, rather than on the server. In essence, the applet allows some functionality to be moved from the server to the client.

The creation of the applet changed Internet programming because it expanded the universe of objects that can move about freely in cyberspace. In general, there are two very broad categories of objects that are transmitted between the server and the client: passive information and dynamic, active programs. For example, when you read your e-mail, you are viewing passive data. Even when you download a program, the program's code is still only passive data until you execute it. By contrast, the applet is a dynamic, self-executing program. Such a program is an active agent on the client computer, yet it is initiated by the server.

As desirable as dynamic, networked programs are, they also present serious problems in the areas of security and portability. Obviously, a program that downloads and executes automatically on the client computer must be prevented from doing harm. It must also be able to run in a variety of different environments and under different operating systems. As you will see, Java solved these problems in an effective and elegant way. Let's look a bit more closely at each.

Security

As you are likely aware, every time you download a “normal” program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources. For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer’s local file system. In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack.

Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer. (You will see how this is accomplished shortly.) The ability to download applets with confidence that no harm will be done and that no security will be breached may have been the single most innovative aspect of Java.

Portability

Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems. For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet. It is not practical to have different versions of the applet for different computers. The *same* code must work on *all* computers. Therefore, some means of generating portable executable code was needed. As you will soon see, the same mechanism that helps ensure security also helps create portability.

Java's Magic: The Bytecode

The key that allows Java to solve both the security and the portability problems just described is that the output of a Java compiler is not executable code. Rather, it is bytecode. *Bytecode* is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the *Java Virtual Machine (JVM)*. In essence, the original JVM was designed as an *interpreter for bytecode*. This may come as a bit of a surprise since many modern languages are designed to be compiled into executable code because of performance concerns. However, the fact that a Java program is executed by the JVM helps solve the major problems associated with web-based programs. Here is why.

Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all understand the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs.

The fact that a Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it can contain the program and prevent it from generating

side effects outside of the system. As you will see, safety is also enhanced by certain restrictions that exist in the Java language.

In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it would run if compiled to executable code. However, with Java, the differential between the two is not so great. Because bytecode has been highly optimized, the use of bytecode enables the JVM to execute programs much faster than you might expect.

Although Java was designed as an interpreted language, there is nothing about Java that prevents on-the-fly compilation of bytecode into native code in order to boost performance. For this reason, the HotSpot technology was introduced not long after Java's initial release. HotSpot provides a Just-In-Time (JIT) compiler for bytecode. When a JIT compiler is part of the JVM, selected portions of bytecode are compiled into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that it is not practical to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be done only at run time. Instead, a JIT compiler compiles code as it is needed, during execution. Furthermore, not all sequences of bytecode are compiled—only those that will benefit from compilation. The remaining code is simply interpreted. However, the just-in-time approach still yields a significant performance boost. Even when dynamic compilation is applied to bytecode, the portability and safety features still apply, because the JVM is still in charge of the execution environment.

Servlets: Java on the Server Side

As useful as applets can be, they are just one half of the client/server equation. Not long after the initial release of Java, it became obvious that Java would also be useful on the server side. The result was the *servlet*. A servlet is a small program that executes on the server. Just as applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server. Thus, with the advent of the servlet, Java spanned both sides of the client/server connection.

Servlets are used to create dynamically generated content that is then served to the client. For example, an online store might use a servlet to look up the price for an item in a database. The price information is then used to dynamically generate a web page that is sent to the browser. Although dynamically generated content is available through mechanisms such as CGI (Common Gateway Interface), the servlet offers several advantages, including increased performance.

Because servlets (like all Java programs) are compiled into bytecode and executed by the JVM, they are highly portable. Thus, the same servlet can be used in a variety of different server environments. The only requirements are that the server support the JVM and a servlet container.

The Java Buzzwords

No discussion of Java's history is complete without a look at the Java buzzwords. Although the fundamental forces that necessitated the invention of Java are portability and security, other factors also played an important role in molding the final form of the language. The key considerations were summed up by the Java team in the following list of buzzwords:

- Simple
- Secure

- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Two of these buzzwords have already been discussed: secure and portable. Let's examine what each of the others implies.

Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.

Object-Oriented

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance nonobjects.

Robust

The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. Many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.

To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional

programming environments. For example, in C/C++, the programmer will often manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because Java provides garbage collection for unused objects.) Exceptional conditions in traditional environments often arise in situations such as division by zero or “file not found,” and they must be managed with clumsy and hard-to-read constructs. Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can—and should—be managed by your program.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java’s easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. At the time of Java’s creation, one of the main problems facing programmers was that no guarantee existed that if you wrote a program today, it would run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was “write once; run anywhere, any time, forever.” To a great extent, this goal was accomplished.

Interpreted and High Performance

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

Distributed

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports *Remote Method Invocation (RMI)*. This feature enables a program to invoke methods across a network.

Dynamic

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

The Evolution of Java

The initial release of Java was nothing short of revolutionary, but it did not mark the end of Java's era of rapid innovation. Unlike most other software systems that usually settle into a pattern of small, incremental improvements, Java continued to evolve at an explosive pace. Soon after the release of Java 1.0, the designers of Java had already created Java 1.1. The features added by Java 1.1 were more significant and substantial than the increase in the minor revision number would have you think. Java 1.1 added many new library elements, redefined the way events are handled, and reconfigured many features of the 1.0 library. It also deprecated (rendered obsolete) several features originally defined by Java 1.0. Thus, Java 1.1 both added to and subtracted from attributes of its original specification.

The next major release of Java was Java 2, where the "2" indicates "second generation." The creation of Java 2 was a watershed event, marking the beginning of Java's "modern age." The first release of Java 2 carried the version number 1.2. It may seem odd that the first release of Java 2 used the 1.2 version number. The reason is that it originally referred to the internal version number of the Java libraries, but then was generalized to refer to the entire release. With Java 2, Sun repackaged the Java product as J2SE (Java 2 Platform Standard Edition), and the version numbers began to be applied to that product.

Java 2 added support for a number of new features, such as Swing and the Collections Framework, and it enhanced the Java Virtual Machine and various programming tools. Java 2 also contained a few deprecations. The most important affected the **Thread** class in which the methods **suspend()**, **resume()**, and **stop()** were deprecated.

J2SE 1.3 was the first major upgrade to the original Java 2 release. For the most part, it added to existing functionality and "tightened up" the development environment. In general, programs written for version 1.2 and those written for version 1.3 are source-code compatible. Although version 1.3 contained a smaller set of changes than the preceding three major releases, it was nevertheless important.

The release of J2SE 1.4 further enhanced Java. This release contained several important upgrades, enhancements, and additions. For example, it added the new keyword **assert**, chained exceptions, and a channel-based I/O subsystem. It also made changes to the Collections Framework and the networking classes. In addition, numerous small changes were made throughout. Despite the significant number of new features, version 1.4 maintained nearly 100 percent source-code compatibility with prior versions.

The next release of Java was J2SE 5, and it was revolutionary. Unlike most of the previous Java upgrades, which offered important, but measured improvements, J2SE 5 fundamentally expanded the scope, power, and range of the language. To grasp the magnitude of the changes that J2SE 5 made to Java, consider the following list of its major new features:

- Generics
- Annotations

- Autoboxing and auto-unboxing
- Enumerations
- Enhanced, for-each style **for** loop
- Variable-length arguments (varargs)
- Static import
- Formatted I/O
- Concurrency utilities

This is not a list of minor tweaks or incremental upgrades. Each item in the list represented a significant addition to the Java language. Some, such as generics, the enhanced **for**, and varargs, introduced new syntax elements. Others, such as autoboxing and auto-unboxing, altered the semantics of the language. Annotations added an entirely new dimension to programming. In all cases, the impact of these additions went beyond their direct effects. They changed the very character of Java itself.

The importance of these new features is reflected in the use of the version number “5.” The next version number for Java would normally have been 1.5. However, the new features were so significant that a shift from 1.4 to 1.5 just didn’t seem to express the magnitude of the change. Instead, Sun elected to increase the version number to 5 as a way of emphasizing that a major event was taking place. Thus, it was named J2SE 5, and the developer’s kit was called JDK 5. However, in order to maintain consistency, Sun decided to use 1.5 as its internal version number, which is also referred to as the *developer version* number. The “5” in J2SE 5 is called the *product version* number.

The next release of Java was called Java SE 6. Sun once again decided to change the name of the Java platform. First, notice that the “2” was dropped. Thus, the platform was now named *Java SE*, and the official product name was *Java Platform, Standard Edition 6*. The Java Development Kit was called JDK 6. As with J2SE 5, the 6 in Java SE 6 is the product version number. The internal, developer version number is 1.6.

Java SE 6 built on the base of J2SE 5, adding incremental improvements. Java SE 6 added no major features to the Java language proper, but it did enhance the API libraries, added several new packages, and offered improvements to the runtime. It also went through several updates during its (in Java terms) long life cycle, with several upgrades added along the way. In general, Java SE 6 served to further solidify the advances made by J2SE 5.

Java SE 7 was the next release of Java, with the Java Development Kit being called JDK 7, and an internal version number of 1.7. Java SE 7 was the first major release of Java since Sun Microsystems was acquired by Oracle. Java SE 7 contained many new features, including significant additions to the language and the API libraries. Upgrades to the Java run-time system that support non-Java languages were also included, but it is the language and library additions that were of most interest to Java programmers.

The new language features were developed as part of *Project Coin*. The purpose of Project Coin was to identify a number of small changes to the Java language that would be incorporated into JDK 7. Although these features were collectively referred to as “small,” the effects of these changes have been quite large in terms of the code they impact. In fact, for

many programmers, these changes may well have been the most important new features in Java SE 7. Here is a list of the language features added by JDK 7:

- A **String** can now control a **switch** statement.
- Binary integer literals.
- Underscores in numeric literals.
- An expanded **try** statement, called *try-with-resources*, that supports automatic resource management. (For example, streams can be closed automatically when they are no longer needed.)
- Type inference (via the *diamond* operator) when constructing a generic instance.
- Enhanced exception handling in which two or more exceptions can be caught by a single **catch** (multi-catch) and better type checking for exceptions that are rethrown.
- Although not a syntax change, the compiler warnings associated with some types of varargs methods were improved, and you have more control over the warnings.

As you can see, even though the Project Coin features were considered small changes to the language, their benefits were much larger than the qualifier “small” would suggest. In particular, the *try-with-resources* statement has profoundly affected the way that stream-based code is written. Also, the ability to use a **String** to control a **switch** statement was a long-desired improvement that simplified coding in many situations.

Java SE 7 made several additions to the Java API library. Two of the most important were the enhancements to the NIO Framework and the addition of the Fork/Join Framework. NIO (which originally stood for *New I/O*) was added to Java in version 1.4. However, the changes added by Java SE 7 fundamentally expanded its capabilities. So significant were the changes, that the term *NIO.2* is often used.

The Fork/Join Framework provides important support for *parallel programming*. Parallel programming is the name commonly given to the techniques that make effective use of computers that contain more than one processor, including multicore systems. The advantage that multicore environments offer is the prospect of significantly increased program performance. The Fork/Join Framework addressed parallel programming by

- Simplifying the creation and use of tasks that can execute concurrently
- Automatically making use of multiple processors

Therefore, by using the Fork/Join Framework, you can easily create scaleable applications that automatically take advantage of the processors available in the execution environment. Of course, not all algorithms lend themselves to parallelization, but for those that do, a significant improvement in execution speed can be obtained.

Java SE 8

The newest release of Java is Java SE 8, with the developer’s kit being called JDK 8. It has an internal version number of 1.8. JDK 8 represents a very significant upgrade to the Java language because of the inclusion of a far-reaching new language feature: the *lambda expression*. The impact of lambda expressions will be profound, changing both the way that

programming solutions are conceptualized and how Java code is written. As explained in detail in Chapter 15, lambda expressions add functional programming features to Java. In the process, lambda expressions can simplify and reduce the amount of source code needed to create certain constructs, such as some types of anonymous classes. The addition of lambda expressions also causes a new operator (the `->`) and a new syntax element to be added to the language. Lambda expressions help ensure that Java will remain the vibrant, nimble language that users have come to expect.

The inclusion of lambda expressions has also had a wide-ranging effect on the Java libraries, with new features being added to take advantage of them. One of the most important is the new stream API, which is packaged in `java.util.stream`. The stream API supports pipeline operations on data and is optimized for lambda expressions. Another very important new package is `java.util.function`. It defines a number of *functional interfaces*, which provide additional support for lambda expressions. Other new lambda-related features are found throughout the API library.

Another lambda-inspired feature affects `interface`. Beginning with JDK 8, it is now possible to define a default implementation for a method specified by an interface. If no implementation for a default method is created, then the default defined by the interface is used. This feature enables interfaces to be gracefully evolved over time because a new method can be added to an interface without breaking existing code. It can also streamline the implementation of an interface when the defaults are appropriate. Other new features in JDK 8 include a new time and date API, type annotations, and the ability to use parallel processing when sorting an array, among others. JDK 8 also bundles support for JavaFX 8, the latest version of Java's new GUI application framework. JavaFX is expected to soon play an important part in nearly all Java applications, ultimately replacing Swing for most GUI-based projects. Part IV of this book provides an introduction to it.

In the final analysis, Java SE 8 is a major release that profoundly expands the capabilities of the language and changes the way that Java code is written. Its effects will be felt throughout the Java universe and for years to come. It truly is that important of a upgrade.

The material in this book has been updated to reflect Java SE 8, with many new features, updates, and additions indicated throughout.

A Culture of Innovation

Since the beginning, Java has been at the center of a culture of innovation. Its original release redefined programming for the Internet. The Java Virtual Machine (JVM) and bytecode changed the way we think about security and portability. The applet (and then the servlet) made the Web come alive. The Java Community Process (JCP) redefined the way that new ideas are assimilated into the language. The world of Java has never stood still for very long. Java SE 8 is the latest release in Java's ongoing, dynamic history.

CHAPTER

2

An Overview of Java

As in all other computer languages, the elements of Java do not exist in isolation. Rather, they work together to form the language as a whole. However, this interrelatedness can make it difficult to describe one aspect of Java without involving several others. Often a discussion of one feature implies prior knowledge of another. For this reason, this chapter presents a quick overview of several key features of Java. The material described here will give you a foothold that will allow you to write and understand simple programs. Most of the topics discussed will be examined in greater detail in the remaining chapters of Part I.

Object-Oriented Programming

Object-oriented programming (OOP) is at the core of Java. In fact, all Java programs are to at least some extent object-oriented. OOP is so integral to Java that it is best to understand its basic principles before you begin writing even simple Java programs. Therefore, this chapter begins with a discussion of the theoretical aspects of OOP.

Two Paradigms

All computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data. That is, some programs are written around “what is happening” and others are written around “who is being affected.” These are the two paradigms that govern how a program is constructed. The first way is called the *process-oriented model*. This approach characterizes a program as a series of linear steps (that is, code). The process-oriented model can be thought of as *code acting on data*. Procedural languages such as C employ this model to considerable success. However, as mentioned in Chapter 1, problems with this approach appear as programs grow larger and more complex.

To manage increasing complexity, the second approach, called *object-oriented programming*, was conceived. Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as *data controlling access to code*. As you will see, by switching the controlling entity to data, you can achieve several organizational benefits.

Abstraction

An essential element of object-oriented programming is *abstraction*. Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. This abstraction allows people to use a car to drive to the grocery store without being overwhelmed by the complexity of the parts that form the car. They can ignore the details of how the engine, transmission, and braking systems work. Instead, they are free to utilize the object as a whole.

A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. From the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units. For instance, the sound system consists of a radio, a CD player, and/or a tape or MP3 player. The point is that you manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions.

Hierarchical abstractions of complex systems can also be applied to computer programs. The data from a traditional process-oriented program can be transformed by abstraction into its component objects. A sequence of process steps can become a collection of messages between these objects. Thus, each of these objects describes its own unique behavior. You can treat these objects as concrete entities that respond to messages telling them to *do something*. This is the essence of object-oriented programming.

Object-oriented concepts form the heart of Java just as they form the basis for human understanding. It is important that you understand how these concepts translate into programs. As you will see, object-oriented programming is a powerful and natural paradigm for creating programs that survive the inevitable changes accompanying the life cycle of any major software project, including conception, growth, and aging. For example, once you have well-defined objects and clean, reliable interfaces to those objects, you can gracefully decommission or replace parts of an older system without fear.

The Three OOP Principles

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. They are encapsulation, inheritance, and polymorphism. Let's take a look at these concepts now.

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. To relate this to the real world, consider the automatic transmission on an automobile. It encapsulates hundreds of bits of information about your engine, such as how much you are accelerating, the pitch of the surface you are on, and the position of the shift lever. You, as the user, have only one method of affecting this complex encapsulation: by moving the gear-shift lever. You can't affect the transmission by using the turn signal or windshield wipers, for example. Thus, the gear-shift lever is a well-defined (indeed, unique) interface to the transmission. Further, what occurs inside the

transmission does not affect objects outside the transmission. For example, shifting gears does not turn on the headlights! Because an automatic transmission is encapsulated, dozens of car manufacturers can implement one in any way they please. However, from the driver's point of view, they all work the same. This same idea can be applied to programming. The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects.

In Java, the basis of encapsulation is the class. Although the class will be examined in great detail later in this book, the following brief discussion will be helpful now. A *class* defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as *instances of a class*. Thus, a class is a logical construct; an object has physical reality.

When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called *members* of the class. Specifically, the data defined by the class are referred to as *member variables* or *instance variables*. The code that operates on that data is referred to as *member methods* or just *methods*. (If you are familiar with C/C++, it may help to know that what a Java programmer calls a *method*, a C/C++ programmer calls a *function*.) In properly written Java programs, the methods define how the member variables can be used. This means that the behavior and interface of a class are defined by the methods that operate on its instance data.

Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked private or public. The *public* interface of a class represents everything that external users of the class need to know, or may know. The *private* methods and data can only be accessed by code that is a member of the class. Therefore, any other code that is not a member of the class cannot access a private method or variable. Since the private members of a class may only be accessed by other parts of your program through the class' public methods, you can ensure that no improper actions take place. Of course, this means that the public interface should be carefully designed not to expose too much of the inner workings of a class (see Figure 2-1).

Inheritance

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. As mentioned earlier, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Golden Retriever is part of the classification *dog*, which in turn is part of the *mammal* class, which is under the larger class *animal*. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. Let's take a closer look at this process.

Most people naturally view the world as made up of objects that are related to each other in a hierarchical way, such as animals, mammals, and dogs. If you wanted to describe animals in an abstract way, you would say they have some attributes, such as size, intelligence, and type of skeletal system. Animals also have certain behavioral aspects; they eat, breathe, and sleep. This description of attributes and behavior is the class definition for animals.

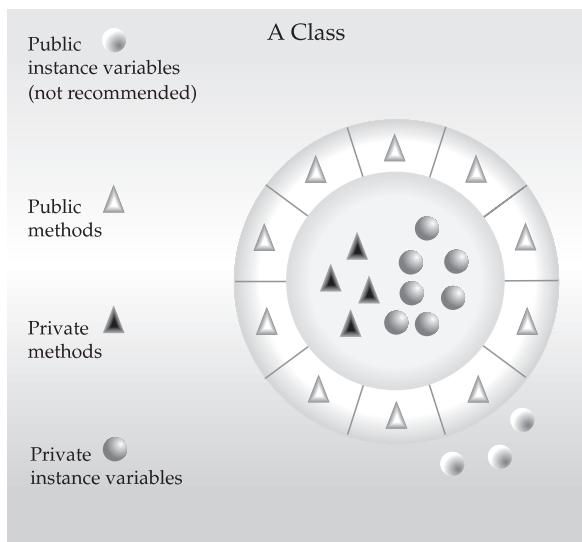
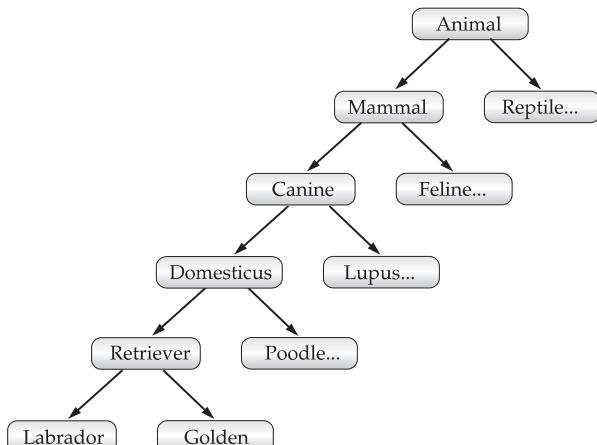


Figure 2-1 Encapsulation: public methods can be used to protect private data.

If you wanted to describe a more specific class of animals, such as mammals, they would have more specific attributes, such as type of teeth and mammary glands. This is known as a *subclass* of animals, where animals are referred to as mammals' *superclass*.

Since mammals are simply more precisely specified animals, they *inherit* all of the attributes from animals. A deeply inherited subclass inherits all of the attributes from each of its ancestors in the *class hierarchy*.

Inheritance interacts with encapsulation as well. If a given class encapsulates some attributes, then any subclass will have the same attributes *plus* any that it adds as part of its specialization (see Figure 2-2). This is a key concept that lets object-oriented programs grow in complexity linearly rather than geometrically. A new subclass inherits all of the attributes of all of its ancestors. It does not have unpredictable interactions with the majority of the rest of the code in the system.



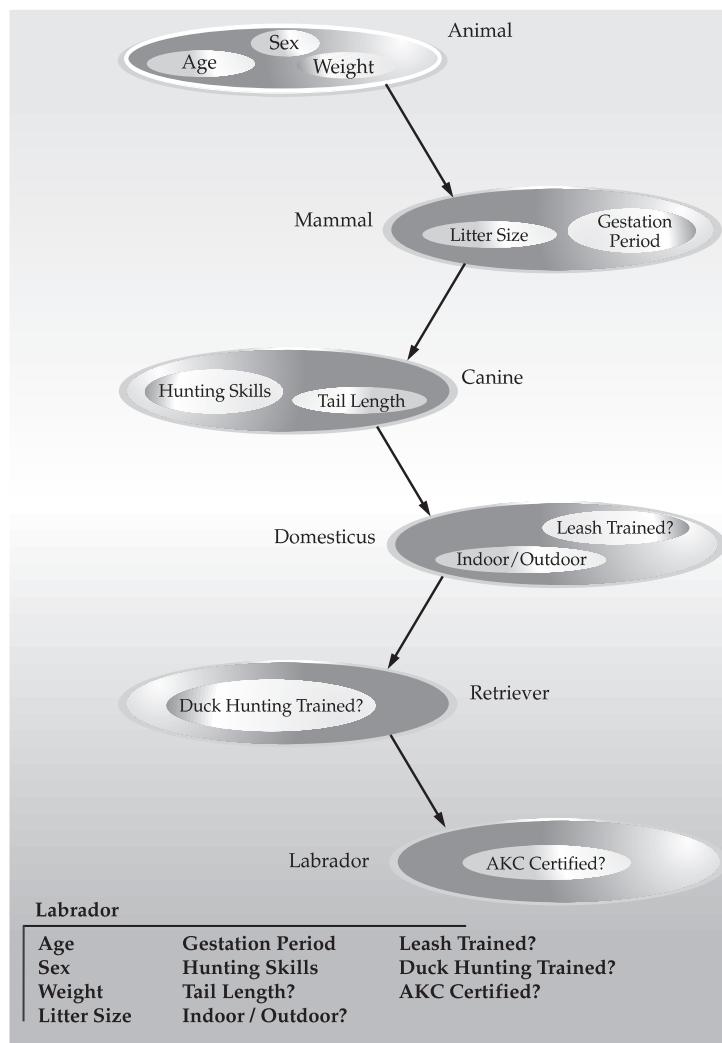


Figure 2-2 Labrador inherits the encapsulation of all its superclasses.

Polymorphism

Polymorphism (from Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. Consider a stack (which is a last-in, first-out list). You might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs. In a non-object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However, because of polymorphism, in Java you can specify a general set of stack routines that all share the same names.

More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.” This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a *general class of action*. It is the compiler’s job to select the *specific action* (that is, method) as it applies to each situation. You, the programmer, do not need to make this selection manually. You need only remember and utilize the general interface.

Extending the dog analogy, a dog’s sense of smell is polymorphic. If the dog smells a cat, it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl. The same sense of smell is at work in both situations. The difference is what is being smelled, that is, the type of data being operated upon by the dog’s nose! This same general concept can be implemented in Java as it applies to methods within a Java program.

Polymorphism, Encapsulation, and Inheritance Work Together

When properly applied, polymorphism, encapsulation, and inheritance combine to produce a programming environment that supports the development of far more robust and scaleable programs than does the process-oriented model. A well-designed hierarchy of classes is the basis for reusing the code in which you have invested time and effort developing and testing. Encapsulation allows you to migrate your implementations over time without breaking the code that depends on the public interface of your classes. Polymorphism allows you to create clean, sensible, readable, and resilient code.

Of the two real-world examples, the automobile more completely illustrates the power of object-oriented design. Dogs are fun to think about from an inheritance standpoint, but cars are more like programs. All drivers rely on inheritance to drive different types (subclasses) of vehicles. Whether the vehicle is a school bus, a Mercedes sedan, a Porsche, or the family minivan, drivers can all more or less find and operate the steering wheel, the brakes, and the accelerator. After a bit of gear grinding, most people can even manage the difference between a stick shift and an automatic, because they fundamentally understand their common superclass, the transmission.

People interface with encapsulated features on cars all the time. The brake and gas pedals hide an incredible array of complexity with an interface so simple you can operate them with your feet! The implementation of the engine, the style of brakes, and the size of the tires have no effect on how you interface with the class definition of the pedals.

The final attribute, polymorphism, is clearly reflected in the ability of car manufacturers to offer a wide array of options on basically the same vehicle. For example, you can get an antilock braking system or traditional brakes, power or rack-and-pinion steering, and 4-, 6-, or 8-cylinder engines. Either way, you will still press the brake pedal to stop, turn the steering wheel to change direction, and press the accelerator when you want to move. The same interface can be used to control a number of different implementations.

As you can see, it is through the application of encapsulation, inheritance, and polymorphism that the individual parts are transformed into the object known as a car. The same is also true of computer programs. By the application of object-oriented principles, the various parts of a complex program can be brought together to form a cohesive, robust, maintainable whole.

As mentioned at the start of this section, every Java program is object-oriented. Or, put more precisely, every Java program involves encapsulation, inheritance, and polymorphism. Although the short example programs shown in the rest of this chapter and in the next few chapters may not seem to exhibit all of these features, they are nevertheless present. As you

will see, many of the features supplied by Java are part of its built-in class libraries, which do make extensive use of encapsulation, inheritance, and polymorphism.

A First Simple Program

Now that the basic object-oriented underpinning of Java has been discussed, let's look at some actual Java programs. Let's start by compiling and running the short sample program shown here. As you will see, this involves a little more work than you might imagine.

```
/*
    This is a simple Java program.
    Call this file "Example.java".
*/
class Example {
    // Your program begins with a call to main().
    public static void main(String args[]) {
        System.out.println("This is a simple Java program.");
    }
}
```

NOTE The descriptions that follow use the standard Java SE 8 Development Kit (JDK 8), which is available from Oracle. If you are using an integrated development environment (IDE), then you will need to follow a different procedure for compiling and executing Java programs. In this case, consult your IDE's documentation for details.

Entering the Program

For most computer languages, the name of the file that holds the source code to a program is immaterial. However, this is not the case with Java. The first thing that you must learn about Java is that the name you give to a source file is very important. For this example, the name of the source file should be **Example.java**. Let's see why.

In Java, a source file is officially called a *compilation unit*. It is a text file that contains (among other things) one or more class definitions. (For now, we will be using source files that contain only one class.) The Java compiler requires that a source file use the **.java** filename extension.

As you can see by looking at the program, the name of the class defined by the program is also **Example**. This is not a coincidence. In Java, all code must reside inside a class. By convention, the name of the main class should match the name of the file that holds the program. You should also make sure that the capitalization of the filename matches the class name. The reason for this is that Java is case-sensitive. At this point, the convention that filenames correspond to class names may seem arbitrary. However, this convention makes it easier to maintain and organize your programs.

Compiling the Program

To compile the **Example** program, execute the compiler, **javac**, specifying the name of the source file on the command line, as shown here:

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

The **javac** compiler creates a file called **Example.class** that contains the bytecode version of the program. As discussed earlier, the Java bytecode is the intermediate representation of

your program that contains instructions the Java Virtual Machine will execute. Thus, the output of **javac** is not code that can be directly executed.

To actually run the program, you must use the Java application launcher called **java**. To do so, pass the class name **Example** as a command-line argument, as shown here:

```
C:\>java Example
```

When the program is run, the following output is displayed:

```
This is a simple Java program.
```

When Java source code is compiled, each individual class is put into its own output file named after the class and using the **.class** extension. This is why it is a good idea to give your Java source files the same name as the class they contain—the name of the source file will match the name of the **.class** file. When you execute **java** as just shown, you are actually specifying the name of the class that you want to execute. It will automatically search for a file by that name that has the **.class** extension. If it finds the file, it will execute the code contained in the specified class.

A Closer Look at the First Sample Program

Although **Example.java** is quite short, it includes several key features that are common to all Java programs. Let's closely examine each part of the program.

The program begins with the following lines:

```
/*
 * This is a simple Java program.
 * Call this file "Example.java".
 */
```

This is a *comment*. Like most other programming languages, Java lets you enter a remark into a program's source file. The contents of a comment are ignored by the compiler. Instead, a comment describes or explains the operation of the program to anyone who is reading its source code. In this case, the comment describes the program and reminds you that the source file should be called **Example.java**. Of course, in real applications, comments generally explain how some part of the program works or what a specific feature does.

Java supports three styles of comments. The one shown at the top of the program is called a *multiline comment*. This type of comment must begin with `/*` and end with `*/`. Anything between these two comment symbols is ignored by the compiler. As the name suggests, a multiline comment may be several lines long.

The next line of code in the program is shown here:

```
class Example {
```

This line uses the keyword **class** to declare that a new class is being defined. **Example** is an *identifier* that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace (`{`) and the closing curly brace (`}`). For the moment, don't worry too much about the details of a class except to note that in Java, all program activity occurs within one. This is one reason why all Java programs are (at least a little bit) object-oriented.

The next line in the program is the *single-line comment*, shown here:

```
// Your program begins with a call to main().
```

This is the second type of comment supported by Java. A *single-line comment* begins with a // and ends at the end of the line. As a general rule, programmers use multiline comments for longer remarks and single-line comments for brief, line-by-line descriptions. The third type of comment, a *documentation comment*, will be discussed in the “Comments” section later in this chapter.

The next line of code is shown here:

```
public static void main(String args[ ]) {
```

This line begins the **main()** method. As the comment preceding it suggests, this is the line at which the program will begin executing. All Java applications begin execution by calling **main()**. The full meaning of each part of this line cannot be given now, since it involves a detailed understanding of Java’s approach to encapsulation. However, since most of the examples in the first part of this book will use this line of code, let’s take a brief look at each part now.

The **public** keyword is an *access modifier*, which allows the programmer to control the visibility of class members. When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared. (The opposite of **public** is **private**, which prevents a member from being used by code defined outside of its class.) In this case, **main()** must be declared as **public**, since it must be called by code outside of its class when the program is started. The keyword **static** allows **main()** to be called without having to instantiate a particular instance of the class. This is necessary since **main()** is called by the Java Virtual Machine before any objects are made. The keyword **void** simply tells the compiler that **main()** does not return a value. As you will see, methods may also return values. If all this seems a bit confusing, don’t worry. All of these concepts will be discussed in detail in subsequent chapters.

As stated, **main()** is the method called when a Java application begins. Keep in mind that Java is case-sensitive. Thus, **Main** is different from **main**. It is important to understand that the Java compiler will compile classes that do not contain a **main()** method. But **java** has no way to run these classes. So, if you had typed **Main** instead of **main**, the compiler would still compile your program. However, **java** would report an error because it would be unable to find the **main()** method.

Any information that you need to pass to a method is received by variables specified within the set of parentheses that follow the name of the method. These variables are called *parameters*. If there are no parameters required for a given method, you still need to include the empty parentheses. In **main()**, there is only one parameter, albeit a complicated one. **String args[]** declares a parameter named **args**, which is an array of instances of the class **String**. (Arrays are collections of similar objects.) Objects of type **String** store character strings. In this case, **args** receives any command-line arguments present when the program is executed. This program does not make use of this information, but other programs shown later in this book will.

The last character on the line is the {. This signals the start of **main()**’s body. All of the code that comprises a method will occur between the method’s opening curly brace and its closing curly brace.

One other point: **main()** is simply a starting place for your program. A complex program will have dozens of classes, only one of which will need to have a **main()** method to get things started. Furthermore, in some cases, you won't need **main()** at all. For example, when creating applets—Java programs that are embedded in web browsers—you won't use **main()** since the web browser uses a different means of starting the execution of applets.

The next line of code is shown here. Notice that it occurs inside **main()**.

```
System.out.println("This is a simple Java program.");
```

This line outputs the string "This is a simple Java program." followed by a new line on the screen. Output is actually accomplished by the built-in **println()** method. In this case, **println()** displays the string which is passed to it. As you will see, **println()** can be used to display other types of information, too. The line begins with **System.out**. While too complicated to explain in detail at this time, briefly, **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console.

As you have probably guessed, console output (and input) is not used frequently in most real-world Java applications. Since most modern computing environments are windowed and graphical in nature, console I/O is used mostly for simple utility programs, demonstration programs, and server-side code. Later in this book, you will learn other ways to generate output using Java. But for now, we will continue to use the console I/O methods.

Notice that the **println()** statement ends with a semicolon. All statements in Java end with a semicolon. The reason that the other lines in the program do not end in a semicolon is that they are not, technically, statements.

The first } in the program ends **main()**, and the last } ends the **Example** class definition.

A Second Short Program

Perhaps no other concept is more fundamental to a programming language than that of a variable. As you may know, a variable is a named memory location that may be assigned a value by your program. The value of a variable may be changed during the execution of the program. The next program shows how a variable is declared and how it is assigned a value. The program also illustrates some new aspects of console output. As the comments at the top of the program state, you should call this file **Example2.java**.

```
/*
Here is another short example.
Call this file "Example2.java".
*/
class Example2 {
    public static void main(String args []) {
        int num; // this declares a variable called num
        num = 100; // this assigns num the value 100
        System.out.println("This is num: " + num);
        num = num * 2;
        System.out.print("The value of num * 2 is ");
    }
}
```

```
        System.out.println(num) ;  
    }  
}
```

When you run this program, you will see the following output:

```
This is num: 100  
The value of num * 2 is 200
```

Let's take a close look at why this output is generated. The first new line in the program is shown here:

```
int num; // this declares a variable called num
```

This line declares an integer variable called **num**. Java (like most other languages) requires that variables be declared before they are used.

Following is the general form of a variable declaration:

```
type var-name;
```

Here, *type* specifies the type of variable being declared, and *var-name* is the name of the variable. If you want to declare more than one variable of the specified type, you may use a comma-separated list of variable names. Java defines several data types, including integer, character, and floating-point. The keyword **int** specifies an integer type.

In the program, the line

```
num = 100; // this assigns num the value 100
```

assigns to **num** the value 100. In Java, the assignment operator is a single equal sign.

The next line of code outputs the value of **num** preceded by the string "This is num:".

```
System.out.println("This is num: " + num);
```

In this statement, the plus sign causes the value of **num** to be appended to the string that precedes it, and then the resulting string is output. (Actually, **num** is first converted from an integer into its string equivalent and then concatenated with the string that precedes it. This process is described in detail later in this book.) This approach can be generalized. Using the + operator, you can join together as many items as you want within a single **println()** statement.

The next line of code assigns **num** the value of **num** times 2. Like most other languages, Java uses the * operator to indicate multiplication. After this line executes, **num** will contain the value 200.

Here are the next two lines in the program:

```
System.out.print ("The value of num * 2 is ");  
System.out.println (num);
```

Several new things are occurring here. First, the built-in method **print()** is used to display the string "The value of num * 2 is ". This string is not followed by a newline. This means that when the next output is generated, it will start on the same line. The **print()** method is just like **println()**, except that it does not output a newline character after each call. Now look at the call to **println()**. Notice that **num** is used by itself. Both **print()** and **println()** can be used to output values of any of Java's built-in types.

Two Control Statements

Although Chapter 5 will look closely at control statements, two are briefly introduced here so that they can be used in example programs in Chapters 3 and 4. They will also help illustrate an important aspect of Java: blocks of code.

The if Statement

The Java **if** statement works much like the IF statement in any other language. Further, it is syntactically identical to the **if** statements in C, C++, and C#. Its simplest form is shown here:

```
if(condition) statement;
```

Here, *condition* is a Boolean expression. If *condition* is true, then the statement is executed. If *condition* is false, then the statement is bypassed. Here is an example:

```
if(num < 100) System.out.println("num is less than 100");
```

In this case, if **num** contains a value that is less than 100, the conditional expression is true, and **println()** will execute. If **num** contains a value greater than or equal to 100, then the **println()** method is bypassed.

As you will see in Chapter 4, Java defines a full complement of relational operators which may be used in a conditional expression. Here are a few:

Operator	Meaning
<	Less than
>	Greater than
==	Equal to

Notice that the test for equality is the double equal sign.

Here is a program that illustrates the **if** statement:

```
/*
Demonstrate the if.

Call this file "IfSample.java".
*/
class IfSample {
    public static void main(String args[]) {
        int x, y;

        x = 10;
        y = 20;

        if(x < y) System.out.println("x is less than y");

        x = x * 2;
        if(x == y) System.out.println("x now equal to y");
    }
}
```

```

x = x * 2;
if(x > y) System.out.println("x now greater than y");

// this won't display anything
if(x == y) System.out.println("you won't see this");
}
}

```

The output generated by this program is shown here:

```

x is less than y
x now equal to y
x now greater than y

```

Notice one other thing in this program. The line

```
int x, y;
```

declares two variables, **x** and **y**, by use of a comma-separated list.

The for Loop

As you may know from your previous programming experience, loop statements are an important part of nearly any programming language. Java is no exception. In fact, as you will see in Chapter 5, Java supplies a powerful assortment of loop constructs. Perhaps the most versatile is the **for** loop. The simplest form of the **for** loop is shown here:

```
for(initialization; condition; iteration) statement;
```

In its most common form, the *initialization* portion of the loop sets a loop control variable to an initial value. The *condition* is a Boolean expression that tests the loop control variable. If the outcome of that test is true, the **for** loop continues to iterate. If it is false, the loop terminates. The *iteration* expression determines how the loop control variable is changed each time the loop iterates. Here is a short program that illustrates the **for** loop:

```

/*
Demonstrate the for loop.

Call this file "ForTest.java".
*/
class ForTest {
    public static void main(String args[]) {
        int x;

        for(x = 0; x<10; x = x+1)
            System.out.println("This is x: " + x);
    }
}

```

This program generates the following output:

```

This is x: 0
This is x: 1
This is x: 2
This is x: 3

```

```
This is x: 4
This is x: 5
This is x: 6
This is x: 7
This is x: 8
This is x: 9
```

In this example, **x** is the loop control variable. It is initialized to zero in the initialization portion of the **for**. At the start of each iteration (including the first one), the conditional test **x < 10** is performed. If the outcome of this test is true, the **println()** statement is executed, and then the iteration portion of the loop is executed, which increases **x** by 1. This process continues until the conditional test is false.

As a point of interest, in professionally written Java programs you will almost never see the iteration portion of the loop written as shown in the preceding program. That is, you will seldom see statements like this:

```
x = x + 1;
```

The reason is that Java includes a special increment operator which performs this operation more efficiently. The increment operator is **++**. (That is, two plus signs back to back.) The increment operator increases its operand by one. By use of the increment operator, the preceding statement can be written like this:

```
++;
```

Thus, the **for** in the preceding program will usually be written like this:

```
for(x = 0; x<10; ++)
```

You might want to try this. As you will see, the loop still runs exactly the same as it did before.

Java also provides a decrement operator, which is specified as **--**. This operator decreases its operand by one.

Using Blocks of Code

Java allows two or more statements to be grouped into *blocks of code*, also called *code blocks*. This is done by enclosing the statements between opening and closing curly braces. Once a block of code has been created, it becomes a logical unit that can be used any place that a single statement can. For example, a block can be a target for Java's **if** and **for** statements. Consider this **if** statement:

```
if(x < y) { // begin a block
    x = y;
    y = 0;
} // end of block
```

Here, if **x** is less than **y**, then both statements inside the block will be executed. Thus, the two statements inside the block form a logical unit, and one statement cannot execute without the other also executing. The key point here is that whenever you need to logically link two or more statements, you do so by creating a block.

Let's look at another example. The following program uses a block of code as the target of a **for** loop.

```
/*
Demonstrate a block of code.

Call this file "BlockTest.java"
*/
class BlockTest {
    public static void main(String args[]) {
        int x, y;

        y = 20;

        // the target of this loop is a block
        for(x = 0; x<10; x++) {
            System.out.println("This is x: " + x);
            System.out.println("This is y: " + y);
            y = y - 2;
        }
    }
}
```

The output generated by this program is shown here:

```
This is x: 0
This is y: 20
This is x: 1
This is y: 18
This is x: 2
This is y: 16
This is x: 3
This is y: 14
This is x: 4
This is y: 12
This is x: 5
This is y: 10
This is x: 6
This is y: 8
This is x: 7
This is y: 6
This is x: 8
This is y: 4
This is x: 9
This is y: 2
```

In this case, the target of the **for** loop is a block of code and not just a single statement. Thus, each time the loop iterates, the three statements inside the block will be executed. This fact is, of course, evidenced by the output generated by the program.

As you will see later in this book, blocks of code have additional properties and uses. However, the main reason for their existence is to create logically inseparable units of code.

Lexical Issues

Now that you have seen several short Java programs, it is time to more formally describe the atomic elements of Java. Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords. The operators are described in the next chapter. The others are described next.

Whitespace

Java is a free-form language. This means that you do not need to follow any special indentation rules. For instance, the **Example** program could have been written all on one line or in any other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator. In Java, whitespace is a space, tab, or newline.

Identifiers

Identifiers are used to name things, such as classes, variables, and methods. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. (The dollar-sign character is not intended for general use.) They must not begin with a number, lest they be confused with a numeric literal. Again, Java is case-sensitive, so **VALUE** is a different identifier than **Value**. Some examples of valid identifiers are

AvgTemp	count	a4	\$test	this_is_ok
---------	-------	----	--------	------------

Invalid identifier names include these:

2count	high-temp	Not/ok
--------	-----------	--------

NOTE Beginning with JDK 8, the use of an underscore by itself as an identifier is not recommended.

Literals

A constant value in Java is created by using a *literal* representation of it. For example, here are some literals:

100	98.6	'X'	"This is a test"
-----	------	-----	------------------

Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

Comments

As mentioned, there are three types of comments defined by Java. You have already seen two: single-line and multiline. The third type is called a *documentation comment*. This type of comment is used to produce an HTML file that documents your program. The

documentation comment begins with a `/**` and ends with a `*/`. Documentation comments are explained in the Appendix.

Separators

In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. As you have seen, it is used to terminate statements. The separators are shown in the following table:

Symbol	Name	Purpose
()	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a <code>for</code> statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.
::	Colons	Used to create a method or constructor reference. (Added by JDK 8.)

The Java Keywords

There are 50 keywords currently defined in the Java language (see Table 2-1). These keywords, combined with the syntax of the operators and separators, form the foundation

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Table 2-1 Java Keywords

of the Java language. These keywords cannot be used as identifiers. Thus, they cannot be used as names for a variable, class, or method.

The keywords **const** and **goto** are reserved but not used. In the early days of Java, several other keywords were reserved for possible future use. However, the current specification for Java defines only the keywords shown in Table 2-1.

In addition to the keywords, Java reserves the following: **true**, **false**, and **null**. These are values defined by Java. You may not use these words for the names of variables, classes, and so on.

The Java Class Libraries

The sample programs shown in this chapter make use of two of Java's built-in methods: **println()** and **print()**. As mentioned, these methods are available through **System.out**. **System** is a class predefined by Java that is automatically included in your programs. In the larger view, the Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics. The standard classes also provide support for a graphical user interface (GUI). Thus, Java as a totality is a combination of the Java language itself, plus its standard classes. As you will see, the class libraries provide much of the functionality that comes with Java. Indeed, part of becoming a Java programmer is learning to use the standard Java classes. Throughout Part I of this book, various elements of the standard library classes and methods are described as needed. In Part II, several class libraries are described in detail.

CHAPTER

3

Data Types, Variables, and Arrays

This chapter examines three of Java's most fundamental elements: data types, variables, and arrays. As with all modern programming languages, Java supports several types of data. You may use these types to declare variables and to create arrays. As you will see, Java's approach to these items is clean, efficient, and cohesive.

Java Is a Strongly Typed Language

It is important to state at the outset that Java is a strongly typed language. Indeed, part of Java's safety and robustness comes from this fact. Let's see what this means. First, every variable has a type, every expression has a type, and every type is strictly defined. Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility. There are no automatic coercions or conversions of conflicting types as in some languages. The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

The Primitive Types

Java defines eight *primitive* types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. The primitive types are also commonly referred to as *simple* types, and both terms will be used in this book. These can be put in four groups:

- **Integers** This group includes **byte**, **short**, **int**, and **long**, which are for whole-valued signed numbers.
- **Floating-point numbers** This group includes **float** and **double**, which represent numbers with fractional precision.
- **Characters** This group includes **char**, which represents symbols in a character set, like letters and numbers.
- **Boolean** This group includes **boolean**, which is a special type for representing true/false values.

You can use these types as-is, or to construct arrays or your own class types. Thus, they form the basis for all other types of data that you can create.

The primitive types represent single values—not complex objects. Although Java is otherwise completely object-oriented, the primitive types are not. They are analogous to the simple types found in most other non-object-oriented languages. The reason for this is efficiency. Making the primitive types into objects would have degraded performance too much.

The primitive types are defined to have an explicit range and mathematical behavior. Languages such as C and C++ allow the size of an integer to vary based upon the dictates of the execution environment. However, Java is different. Because of Java's portability requirement, all data types have a strictly defined range. For example, an **int** is always 32 bits, regardless of the particular platform. This allows programs to be written that are guaranteed to run *without porting* on any machine architecture. While strictly specifying the size of an integer may cause a small loss of performance in some environments, it is necessary in order to achieve portability.

Let's look at each type of data in turn.

I ntegers

Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers. Many other computer languages support both signed and unsigned integers. However, Java's designers felt that unsigned integers were unnecessary. Specifically, they felt that the concept of *unsigned* was used mostly to specify the behavior of the *high-order bit*, which defines the *sign* of an integer value. As you will see in Chapter 4, Java manages the meaning of the high-order bit differently, by adding a special “unsigned right shift” operator. Thus, the need for an unsigned integer type was eliminated.

The *width* of an integer type should not be thought of as the amount of storage it consumes, but rather as the *behavior* it defines for variables and expressions of that type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them. The width and ranges of these integer types vary widely, as shown in this table:

Name	Width	Range
long	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	-2,147,483,648 to 2,147,483,647
short	16	-32,768 to 32,767
byte	8	-128 to 127

Let's look at each type of integer.

byte

The smallest integer type is **byte**. This is a signed 8-bit type that has a range from -128 to 127. Variables of type **byte** are especially useful when you're working with a stream of data from a network or file. They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types.

Byte variables are declared by use of the **byte** keyword. For example, the following declares two **byte** variables called **b** and **c**:

```
byte b, c;
```

short

short is a signed 16-bit type. It has a range from $-32,768$ to $32,767$. It is probably the least-used Java type. Here are some examples of **short** variable declarations:

```
short s;
short t;
```

int

The most commonly used integer type is **int**. It is a signed 32-bit type that has a range from $-2,147,483,648$ to $2,147,483,647$. In addition to other uses, variables of type **int** are commonly employed to control loops and to index arrays. Although you might think that using a **byte** or **short** would be more efficient than using an **int** in situations in which the larger range of an **int** is not needed, this may not be the case. The reason is that when **byte** and **short** values are used in an expression, they are *promoted* to **int** when the expression is evaluated. (Type promotion is described later in this chapter.) Therefore, **int** is often the best choice when an integer is needed.

long

long is a signed 64-bit type and is useful for those occasions where an **int** type is not large enough to hold the desired value. The range of a **long** is quite large. This makes it useful when big, whole numbers are needed. For example, here is a program that computes the number of miles that light will travel in a specified number of days:

```
// Compute distance light travels using long variables.
class Light {
    public static void main(String args[]) {
        int lightspeed;
        long days;
        long seconds;
        long distance;

        // approximate speed of light in miles per second
        lightspeed = 186000;

        days = 1000; // specify number of days here

        seconds = days * 24 * 60 * 60; // convert to seconds

        distance = lightspeed * seconds; // compute distance

        System.out.print("In " + days);
        System.out.print(" days light will travel about ");
        System.out.println(distance + " miles.");
    }
}
```

This program generates the following output:

```
In 1000 days light will travel about 16070400000000 miles.
```

Clearly, the result could not have been held in an **int** variable.

Floating-Point Types

Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendentals such as sine and cosine, result in a value whose precision requires a floating-point type. Java implements the standard (IEEE-754) set of floating-point types and operators. There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

Name	Width in Bits	Approximate Range
double	64	4.9e-324 to 1.8e+308
float	32	1.4e-045 to 3.4e+038

Each of these floating-point types is examined next.

float

The type **float** specifies a *single-precision* value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type **float** are useful when you need a fractional component, but don't require a large degree of precision. For example, **float** can be useful when representing dollars and cents.

Here are some example **float** variable declarations:

```
float hightemp, lowtemp;
```

double

Double precision, as denoted by the **double** keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as **sin()**, **cos()**, and **sqrt()**, return **double** values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, **double** is the best choice.

Here is a short program that uses **double** variables to compute the area of a circle:

```
// Compute the area of a circle.
class Area {
    public static void main(String args[]) {
        double pi, r, a;

        r = 10.8; // radius of circle
        pi = 3.1416; // pi, approximately
```

```
a = pi * r * r; // compute area  
System.out.println("Area of circle is " + a);  
}  
}
```

Characters

In Java, the data type used to store characters is **char**. However, C/C++ programmers beware: **char** in Java is not the same as **char** in C or C++. In C/C++, **char** is 8 bits wide. This is *not* the case in Java. Instead, Java uses *Unicode* to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. At the time of Java's creation, Unicode required 16 bits. Thus, in Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536. There are no negative **chars**. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255. Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters. Of course, the use of Unicode is somewhat inefficient for languages such as English, German, Spanish, or French, whose characters can easily be contained within 8 bits. But such is the price that must be paid for global portability.

NOTE More information about Unicode can be found at <http://www.unicode.org>.

Here is a program that demonstrates **char** variables:

```
// Demonstrate char data type.  
class CharDemo {  
    public static void main(String args[]) {  
        char ch1, ch2;  
  
        ch1 = 88; // code for X  
        ch2 = 'Y';  
  
        System.out.print("ch1 and ch2: ");  
        System.out.println(ch1 + " " + ch2);  
    }  
}
```

This program displays the following output:

```
ch1 and ch2: X Y
```

Notice that **ch1** is assigned the value 88, which is the ASCII (and Unicode) value that corresponds to the letter X. As mentioned, the ASCII character set occupies the first 127 values in the Unicode character set. For this reason, all the “old tricks” that you may have used with characters in other languages will work in Java, too.

Although **char** is designed to hold Unicode characters, it can also be used as an integer type on which you can perform arithmetic operations. For example, you can add two characters together, or increment the value of a character variable. Consider the following program:

```
// char variables behave like integers.
class CharDemo2 {
    public static void main(String args[]) {
        char ch1;

        ch1 = 'X';
        System.out.println("ch1 contains " + ch1);

        ch1++; // increment ch1
        System.out.println("ch1 is now " + ch1);
    }
}
```

The output generated by this program is shown here:

```
ch1 contains X
ch1 is now Y
```

In the program, **ch1** is first given the value *X*. Next, **ch1** is incremented. This results in **ch1** containing *Y*, the next character in the ASCII (and Unicode) sequence.

NOTE In the formal specification for Java, **char** is referred to as an *integral* type, which means that it is in the same general category as **int**, **short**, **long**, and **byte**. However, because its principal use is for representing Unicode characters, **char** is commonly considered to be in a category of its own.

Booleans

Java has a primitive type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**. This is the type returned by all relational operators, as in the case of **a < b**. **boolean** is also the type *required* by the conditional expressions that govern the control statements such as **if** and **for**.

Here is a program that demonstrates the **boolean** type:

```
// Demonstrate boolean values.
class BoolTest {
    public static void main(String args[]) {
        boolean b;

        b = false;
        System.out.println("b is " + b);
        b = true;
        System.out.println("b is " + b);

        // a boolean value can control the if statement
        if(b) System.out.println("This is executed.");

        b = false;
```

```
if(b) System.out.println("This is not executed.");

// outcome of a relational operator is a boolean value
System.out.println("10 > 9 is " + (10 > 9));
}
}
```

The output generated by this program is shown here:

```
b is false
b is true
This is executed.
10 > 9 is true
```

There are three interesting things to notice about this program. First, as you can see, when a **boolean** value is output by **println()**, "true" or "false" is displayed. Second, the value of a **boolean** variable is sufficient, by itself, to control the **if** statement. There is no need to write an **if** statement like this:

```
if(b == true) ...
```

Third, the outcome of a relational operator, such as **<**, is a **boolean** value. This is why the expression **10>9** displays the value "true." Further, the extra set of parentheses around **10>9** is necessary because the **+** operator has a higher precedence than the **>**.

A Closer Look at Literals

Literals were mentioned briefly in Chapter 2. Now that the built-in types have been formally described, let's take a closer look at them.

Integer Literals

Integers are probably the most commonly used type in the typical program. Any whole number value is an integer literal. Examples are 1, 2, 3, and 42. These are all decimal values, meaning they are describing a base 10 number. Two other bases that can be used in integer literals are *octal* (base eight) and *hexadecimal* (base 16). Octal values are denoted in Java by a leading zero. Normal decimal numbers cannot have a leading zero. Thus, the seemingly valid value 09 will produce an error from the compiler, since 9 is outside of octal's 0 to 7 range. A more common base for numbers used by programmers is hexadecimal, which matches cleanly with modulo 8 word sizes, such as 8, 16, 32, and 64 bits. You signify a hexadecimal constant with a leading zero-x, (**0x** or **0X**). The range of a hexadecimal digit is 0 to 15, so *A* through *F* (or *a* through *f*) are substituted for 10 through 15.

Integer literals create an **int** value, which in Java is a 32-bit integer value. Since Java is strongly typed, you might be wondering how it is possible to assign an integer literal to one of Java's other integer types, such as **byte** or **long**, without causing a type mismatch error. Fortunately, such situations are easily handled. When a literal value is assigned to a **byte** or **short** variable, no error is generated if the literal value is within the range of the target type. An integer literal can always be assigned to a **long** variable. However, to specify a **long** literal, you will need to explicitly tell the compiler that the literal value is of type **long**. You do this by appending an upper- or lowercase *L* to the literal. For example, **0x7fffffffffffffL**

or 9223372036854775807L is the largest **long**. An integer can also be assigned to a **char** as long as it is within range.

Beginning with JDK 7, you can also specify integer literals using binary. To do so, prefix the value with **0b** or **0B**. For example, this specifies the decimal value 10 using a binary literal:

```
int x = 0b1010;
```

Among other uses, the addition of binary literals makes it easier to enter values used as bitmasks. In such a case, the decimal (or hexadecimal) representation of the value does not visually convey its meaning relative to its use. The binary literal does.

Also beginning with JDK 7, you can embed one or more underscores in an integer literal. Doing so makes it easier to read large integer literals. When the literal is compiled, the underscores are discarded. For example, given

```
int x = 123_456_789;
```

the value given to **x** will be 123,456,789. The underscores will be ignored. Underscores can only be used to separate digits. They cannot come at the beginning or the end of a literal. It is, however, permissible for more than one underscore to be used between two digits. For example, this is valid:

```
int x = 123____456____789;
```

The use of underscores in an integer literal is especially useful when encoding such things as telephone numbers, customer ID numbers, part numbers, and so on. They are also useful for providing visual groupings when specifying binary literals. For example, binary values are often visually grouped in four-digits units, as shown here:

```
int x = 0b1101_0101_0001_1010;
```

Floating-Point Literals

Floating-point numbers represent decimal values with a fractional component. They can be expressed in either standard or scientific notation. *Standard notation* consists of a whole number component followed by a decimal point followed by a fractional component. For example, 2.0, 3.14159, and 0.6667 represent valid standard-notation floating-point numbers. *Scientific notation* uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied. The exponent is indicated by an *E* or *e* followed by a decimal number, which can be positive or negative. Examples include 6.022E23, 314159E-05, and 2e+100.

Floating-point literals in Java default to **double** precision. To specify a **float** literal, you must append an *F* or *f* to the constant. You can also explicitly specify a **double** literal by appending a *D* or *d*. Doing so is, of course, redundant. The default **double** type consumes 64 bits of storage, while the smaller **float** type requires only 32 bits.

Hexadecimal floating-point literals are also supported, but they are rarely used. They must be in a form similar to scientific notation, but a **P** or **p**, rather than an **E** or **e**, is used. For example, 0x12.2P2 is a valid floating-point literal. The value following the **P**, called the

binary exponent, indicates the power-of-two by which the number is multiplied. Therefore, **0x12.2P2** represents 72.5.

Beginning with JDK 7, you can embed one or more underscores in a floating-point literal. This feature works the same as it does for integer literals, which were just described. Its purpose is to make it easier to read large floating-point literals. When the literal is compiled, the underscores are discarded. For example, given

```
double num = 9_423_497_862.0;
```

the value given to **num** will be 9,423,497,862.0. The underscores will be ignored. As is the case with integer literals, underscores can only be used to separate digits. They cannot come at the beginning or the end of a literal. It is, however, permissible for more than one underscore to be used between two digits. It is also permissible to use underscores in the fractional portion of the number. For example,

```
double num = 9_423_497.1_0_9;
```

is legal. In this case, the fractional part is **.109**.

Boolean Literals

Boolean literals are simple. There are only two logical values that a **boolean** value can have, **true** and **false**. The values of **true** and **false** do not convert into any numerical representation. The **true** literal in Java does not equal 1, nor does the **false** literal equal 0. In Java, the Boolean literals can only be assigned to variables declared as **boolean** or used in expressions with Boolean operators.

Character Literals

Characters in Java are indices into the Unicode character set. They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators. A literal character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as '*a*', '*z*', and '*@*'. For characters that are impossible to enter directly, there are several escape sequences that allow you to enter the character you need, such as '\'' for the single-quote character itself and '\n' for the newline character. There is also a mechanism for directly entering the value of a character in octal or hexadecimal. For octal notation, use the backslash followed by the three-digit number. For example, '\141' is the letter '*a*'. For hexadecimal, you enter a backslash-u ('\u), then exactly four hexadecimal digits. For example, '\u0061' is the ISO-Latin-1 '*a*' because the top byte is zero. '\ua432' is a Japanese Katakana character. Table 3-1 shows the character escape sequences.

String Literals

String literals in Java are specified like they are in most other languages—by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are

Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
'	Single quote
"	Double quote
\\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

Table 3-1 Character Escape Sequences

```
"Hello World"
"two\nlines"
"\\"This is in quotes\\""
```

The escape sequences and octal/hexadecimal notations that were defined for character literals work the same way inside of string literals. One important thing to note about Java strings is that they must begin and end on the same line. There is no line-continuation escape sequence as there is in some other languages.

NOTE As you may know, in some other languages, including C/C++, strings are implemented as arrays of characters. However, this is not the case in Java. Strings are actually object types. As you will see later in this book, because Java implements strings as objects, Java includes extensive string-handling capabilities that are both powerful and easy to use.

Variables

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime. These elements are examined next.

Declaring a Variable

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

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

Here, *type* is one of Java's atomic types, or the name of a class or interface. (Class and interface types are discussed later in Part I of this book.) The *identifier* is the name of the variable. You can initialize the variable by specifying an equal sign and a value. Keep in mind that the initialization expression must result in a value of the same (or compatible)

type as that specified for the variable. To declare more than one variable of the specified type, use a comma-separated list.

Here are several examples of variable declarations of various types. Note that some include an initialization.

```
int a, b, c;           // declares three ints, a, b, and c.
int d = 3, e, f = 5;   // declares three more ints, initializing
                      // d and f.
byte z = 22;           // initializes z.
double pi = 3.14159;  // declares an approximation of pi.
char x = 'x';          // the variable x has the value 'x'.
```

The identifiers that you choose have nothing intrinsic in their names that indicates their type. Java allows any properly formed identifier to have any declared type.

Dynamic Initialization

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

```
// Demonstrate dynamic initialization.
class DynInit {
    public static void main(String args[]) {
        double a = 3.0, b = 4.0;

        // c is dynamically initialized
        double c = Math.sqrt(a * a + b * b);

        System.out.println("Hypotenuse is " + c);
    }
}
```

Here, three local variables—**a**, **b**, and **c**—are declared. The first two, **a** and **b**, are initialized by constants. However, **c** is initialized dynamically to the length of the hypotenuse (using the Pythagorean theorem). The program uses another of Java’s built-in methods, **sqrt()**, which is a member of the **Math** class, to compute the square root of its argument. The key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

The Scope and Lifetime of Variables

So far, all of the variables used have been declared at the start of the **main()** method. However, Java allows variables to be declared within any block. As explained in Chapter 2, a block is begun with an opening curly brace and ended by a closing curly brace. A block defines a *scope*. Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

Many other computer languages define two general categories of scopes: global and local. However, these traditional scopes do not fit well with Java's strict, object-oriented model. While it is possible to create what amounts to being a global scope, it is by far the exception, not the rule. In Java, the two major scopes are those defined by a class and those defined by a method. Even this distinction is somewhat artificial. However, since the class scope has several unique properties and attributes that do not apply to the scope defined by a method, this distinction makes some sense. Because of the differences, a discussion of class scope (and variables declared within it) is deferred until Chapter 6, when classes are described. For now, we will only examine the scopes defined by or within a method.

The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope. Although this book will look more closely at parameters in Chapter 6, for the sake of this discussion, they work the same as any other method variable.

As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. Indeed, the scope rules provide the foundation for encapsulation.

Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.

To understand the effect of nested scopes, consider the following program:

```
// Demonstrate block scope.
class Scope {
    public static void main(String args[]) {
        int x; // known to all code within main

        x = 10;
        if(x == 10) { // start new scope
            int y = 20; // known only to this block

            // x and y both known here.
            System.out.println("x and y: " + x + " " + y);
            x = y * 2;
        }
        // y = 100; // Error! y not known here

        // x is still known here.
        System.out.println("x is " + x);
    }
}
```

As the comments indicate, the variable **x** is declared at the start of **main()**'s scope and is accessible to all subsequent code within **main()**. Within the **if** block, **y** is declared. Since a block defines a scope, **y** is only visible to other code within its block. This is why outside of its block, the line **y = 100;** is commented out. If you remove the leading comment symbol, a compile-time error will occur, because **y** is not visible outside of its block. Within the **if** block, **x** can be used because code within a block (that is, a nested scope) has access to variables declared by an enclosing scope.

Within a block, variables can be declared at any point, but are valid only after they are declared. Thus, if you define a variable at the start of a method, it is available to all of the code within that method. Conversely, if you declare a variable at the end of a block, it is effectively useless, because no code will have access to it. For example, this fragment is invalid because **count** cannot be used prior to its declaration:

```
// This fragment is wrong!
count = 100; // oops! cannot use count before it is declared!
int count;
```

Here is another important point to remember: variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.

If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered. For example, consider the next program:

```
// Demonstrate lifetime of a variable.
class LifeTime {
    public static void main(String args[]) {
        int x;

        for(x = 0; x < 3; x++) {
            int y = -1; // y is initialized each time block is entered
            System.out.println("y is: " + y); // this always prints -1
            y = 100;
            System.out.println("y is now: " + y);
        }
    }
}
```

The output generated by this program is shown here:

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

As you can see, **y** is reinitialized to **-1** each time the inner **for** loop is entered. Even though it is subsequently assigned the value **100**, this value is lost.

One last point: Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope. For example, the following program is illegal:

```
// This program will not compile
class ScopeErr {
    public static void main(String args[]) {
        int bar = 1;
```

```
{           // creates a new scope
    int bar = 2; // Compile-time error - bar already defined!
}
}
```

Type Conversion and Casting

If you have previous programming experience, then you already know that it is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an **int** value to a **long** variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from **double** to **byte**. Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a *cast*, which performs an explicit conversion between incompatible types. Let's look at both automatic type conversions and casting.

Java's Automatic Conversions

When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:

- The two types are compatible.
 - The destination type is larger than the source type.

When these two conditions are met, a *widening conversion* takes place. For example, the `int` type is always large enough to hold all valid `byte` values, so no explicit cast statement is required.

For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to **char** or **boolean**. Also, **char** and **boolean** are not compatible with each other.

As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type **byte**, **short**, **long**, or **char**.

Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you want to assign an **int** value to a **byte** variable? This conversion will not be performed automatically, because a **byte** is smaller than an **int**. This kind of conversion is sometimes called a *narrowing conversion*, since you are explicitly making the value narrower so that it will fit into the target type.

To create a conversion between two incompatible types, you must use a cast. A *cast* is simply an explicit type conversion. It has this general form:

(*target-type*) *value*

Here, *target-type* specifies the desired type to convert the specified value to. For example, the following fragment casts an **int** to a **byte**. If the integer's value is larger than the range of a **byte**, it will be reduced modulo (the remainder of an integer division by the) **byte**'s range.

```
int a;
byte b;
// ...
b = (byte) a;
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: *truncation*. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

The following program demonstrates some type conversions that require casts:

```
// Demonstrate casts.
class Conversion {
    public static void main(String args[]) {
        byte b;
        int i = 257;
        double d = 323.142;

        System.out.println("\nConversion of int to byte.");
        b = (byte) i;
        System.out.println("i and b " + i + " " + b);

        System.out.println("\nConversion of double to int.");
        i = (int) d;
        System.out.println("d and i " + d + " " + i);

        System.out.println("\nConversion of double to byte.");
        b = (byte) d;
        System.out.println("d and b " + d + " " + b);
    }
}
```

This program generates the following output:

```
Conversion of int to byte.
i and b 257 1
```

```
Conversion of double to int.
d and i 323.142 323
```

```
Conversion of double to byte.
d and b 323.142 67
```

Let's look at each conversion. When the value 257 is cast into a **byte** variable, the result is the remainder of the division of 257 by 256 (the range of a **byte**), which is 1 in this case. When

the **d** is converted to an **int**, its fractional component is lost. When **d** is converted to a **byte**, its fractional component is lost, *and* the value is reduced modulo 256, which in this case is 67.

Automatic Type Promotion in Expressions

In addition to assignments, there is another place where certain type conversions may occur: in expressions. To see why, consider the following. In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand. For example, examine the following expression:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

The result of the intermediate term **a** * **b** easily exceeds the range of either of its **byte** operands. To handle this kind of problem, Java automatically promotes each **byte**, **short**, or **char** operand to **int** when evaluating an expression. This means that the subexpression **a*****b** is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression, **50** * **40**, is legal even though **a** and **b** are both specified as type **byte**.

As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

The code is attempting to store **50** * **2**, a perfectly valid **byte** value, back into a **byte** variable. However, because the operands were automatically promoted to **int** when the expression was evaluated, the result has also been promoted to **int**. Thus, the result of the expression is now of type **int**, which cannot be assigned to a **byte** without the use of a cast. This is true even if, as in this particular case, the value being assigned would still fit in the target type.

In cases where you understand the consequences of overflow, you should use an explicit cast, such as

```
byte b = 50;
b = (byte)(b * 2);
```

which yields the correct value of 100.

The Type Promotion Rules

Java defines several *type promotion* rules that apply to expressions. They are as follows: First, all **byte**, **short**, and **char** values are promoted to **int**, as just described. Then, if one operand is a **long**, the whole expression is promoted to **long**. If one operand is a **float**, the entire expression is promoted to **float**. If any of the operands are **double**, the result is **double**.

The following program demonstrates how each value in the expression gets promoted to match the second argument to each binary operator:

```

class Promote {
    public static void main(String args[]) {
        byte b = 42;
        char c = 'a';
        short s = 1024;
        int i = 50000;
        float f = 5.67f;
        double d = .1234;
        double result = (f * b) + (i / c) - (d * s);
        System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
        System.out.println("result = " + result);
    }
}

```

Let's look closely at the type promotions that occur in this line from the program:

```
double result = (f * b) + (i / c) - (d * s);
```

In the first subexpression, **f * b**, **b** is promoted to a **float** and the result of the subexpression is **float**. Next, in the subexpression **i/c**, **c** is promoted to **int**, and the result is of type **int**. Then, in **d * s**, the value of **s** is promoted to **double**, and the type of the subexpression is **double**. Finally, these three intermediate values, **float**, **int**, and **double**, are considered. The outcome of **float** plus an **int** is a **float**. Then the resultant **float** minus the last **double** is promoted to **double**, which is the type for the final result of the expression.

Arrays

An *array* is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

NOTE If you are familiar with C/C++, be careful. Arrays in Java work differently than they do in those languages.

One-Dimensional Arrays

A *one-dimensional array* is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is

```
type var-name[ ];
```

Here, *type* declares the element type (also called the base type) of the array. The element type determines the data type of each element that comprises the array. Thus, the element type for the array determines what type of data the array will hold. For example, the following declares an array named **month_days** with the type “array of int”:

```
int month_days[];
```

Although this declaration establishes the fact that **month_days** is an array variable, no array actually exists. To link **month_days** with an actual, physical array of integers, you must allocate one using **new** and assign it to **month_days**. **new** is a special operator that allocates memory.

You will look more closely at **new** in a later chapter, but you need to use it now to allocate memory for arrays. The general form of **new** as it applies to one-dimensional arrays appears as follows:

```
array-var = new type [size];
```

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *array-var* is the array variable that is linked to the array. That is, to use **new** to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by **new** will automatically be initialized to zero (for numeric types), **false** (for **boolean**), or **null** (for reference types, which are described in a later chapter). This example allocates a 12-element array of integers and links them to **month_days**:

```
month_days = new int[12];
```

After this statement executes, **month_days** will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.

Let's review: Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using **new**, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated. If the concept of dynamic allocation is unfamiliar to you, don't worry. It will be described at length later in this book.

Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero. For example, this statement assigns the value 28 to the second element of **month_days**:

```
month_days[1] = 28;
```

The next line displays the value stored at index 3:

```
System.out.println(month_days[3]);
```

Putting together all the pieces, here is a program that creates an array of the number of days in each month:

```
// Demonstrate a one-dimensional array.
class Array {
    public static void main(String args[]) {
        int month_days[];
        month_days = new int[12];
        month_days[0] = 31;
        month_days[1] = 28;
        month_days[2] = 31;
        month_days[3] = 30;
        month_days[4] = 31;
        month_days[5] = 30;
```

```

month_days[6] = 31;
month_days[7] = 31;
month_days[8] = 30;
month_days[9] = 31;
month_days[10] = 30;
month_days[11] = 31;
System.out.println("April has " + month_days[3] + " days.");
}
}

```

When you run this program, it prints the number of days in April. As mentioned, Java array indexes start with zero, so the number of days in April is **month_days[3]** or 30.

It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:

```
int month_days[] = new int[12];
```

This is the way that you will normally see it done in professionally written Java programs.

Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An *array initializer* is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use **new**. For example, to store the number of days in each month, the following code creates an initialized array of integers:

```

// An improved version of the previous program.
class AutoArray {
    public static void main(String args[]) {
        int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
                            30, 31 };
        System.out.println("April has " + month_days[3] + " days.");
    }
}

```

When you run this program, you see the same output as that generated by the previous version.

Java strictly checks to make sure you do not accidentally try to store or reference values outside of the range of the array. The Java run-time system will check to be sure that all array indexes are in the correct range. For example, the run-time system will check the value of each index into **month_days** to make sure that it is between 0 and 11 inclusive. If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), you will cause a run-time error.

Here is one more example that uses a one-dimensional array. It finds the average of a set of numbers.

```

// Average an array of values.
class Average {
    public static void main(String args[]) {
        double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
        double result = 0;
        int i;
    }
}

```

```

        for(i=0; i<5; i++)
            result = result + nums[i];
        System.out.println("Average is " + result / 5);
    }
}

```

Multidimensional Arrays

In Java, *multidimensional arrays* are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called **twoD**:

```
int twoD[] [] = new int [4] [5];
```

This allocates a 4 by 5 array and assigns it to **twoD**. Internally, this matrix is implemented as an *array of arrays of int*. Conceptually, this array will look like the one shown in Figure 3-1.

The following program numbers each element in the array from left to right, top to bottom, and then displays these values:

```

// Demonstrate a two-dimensional array.
class TwoDArray {
    public static void main(String args[]) {
        int twoD[] [] = new int [4] [5];
        int i, j, k = 0;

        for(i=0; i<4; i++)
            for(j=0; j<5; j++) {
                twoD[i] [j] = k;
                k++;
            }

        for(i=0; i<4; i++) {
            for(j=0; j<5; j++)
                System.out.print(twoD[i] [j] + " ");
            System.out.println();
        }
    }
}

```

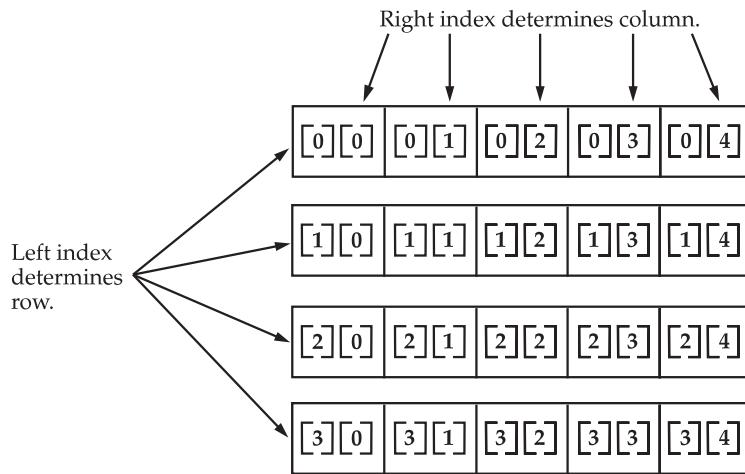
This program generates the following output:

```

0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19

```

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions



```
Given: int twoD [ ] [ ] = new int [4] [5];
```

Figure 3-1 A conceptual view of a 4 by 5, two-dimensional array

separately. For example, this following code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension manually.

```
int twoD[] [] = new int [4] [];
twoD[0] = new int [5];
twoD[1] = new int [5];
twoD[2] = new int [5];
twoD[3] = new int [5];
```

While there is no advantage to individually allocating the second dimension arrays in this situation, there may be in others. For example, when you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension. As stated earlier, since multidimensional arrays are actually arrays of arrays, the length of each array is under your control. For example, the following program creates a two-dimensional array in which the sizes of the second dimension are unequal:

```
// Manually allocate differing size second dimensions.
class TwoDAgain {
    public static void main(String args[]) {
        int twoD[] [] = new int [4] [];
        twoD[0] = new int [1];
        twoD[1] = new int [2];
        twoD[2] = new int [3];
        twoD[3] = new int [4];

        int i, j, k = 0;

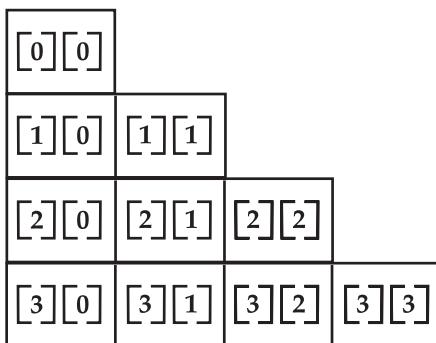
        for(i=0; i<4; i++)
            for(j=0; j<i+1; j++) {
                twoD[i] [j] = k;
                k++;
            }
    }
}
```

```
        }  
  
    for(i=0; i<4; i++) {  
        for(j=0; j<i+1; j++)  
            System.out.print(twoD[i][j] + " ");  
        System.out.println();  
    }  
}
```

This program generates the following output:

0			
1	2		
3	4	5	
6	7	8	9

The array created by this program looks like this:



The use of uneven (or irregular) multidimensional arrays may not be appropriate for many applications, because it runs contrary to what people expect to find when a multidimensional array is encountered. However, irregular arrays can be used effectively in some situations. For example, if you need a very large two-dimensional array that is sparsely populated (that is, one in which not all of the elements will be used), then an irregular array might be a perfect solution.

It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension's initializer within its own set of curly braces. The following program creates a matrix where each element contains the product of the row and column indexes. Also notice that you can use expressions as well as literal values inside of array initializers.

```
// Initialize a two-dimensional array.  
class Matrix {  
    public static void main(String args[]) {  
        double m[][] = {  
            { 0*0, 1*0, 2*0, 3*0 },  
            { 0*1, 1*1, 2*1, 3*1 },  
            { 0*2, 1*2, 2*2, 3*2 },  
            { 0*3, 1*3, 2*3, 3*3 }  
        };  
    }  
}
```

```
    };
    int i, j;

    for(i=0; i<4; i++) {
        for(j=0; j<4; j++)
            System.out.print(m[i][j] + " ");
        System.out.println();
    }
}
```

When you run this program, you will get the following output:

```
0.0 0.0 0.0 0.0
0.0 1.0 2.0 3.0
0.0 2.0 4.0 6.0
0.0 3.0 6.0 9.0
```

As you can see, each row in the array is initialized as specified in the initialization lists.

Let's look at one more example that uses a multidimensional array. The following program creates a 3 by 4 by 5, three-dimensional array. It then loads each element with the product of its indexes. Finally, it displays these products.

```
// Demonstrate a three-dimensional array.
class ThreeDMatrix {
    public static void main(String args[]) {
        int threeD[][][] = new int[3][4][5];
        int i, j, k;

        for(i=0; i<3; i++)
            for(j=0; j<4; j++)
                for(k=0; k<5; k++)
                    threeD[i][j][k] = i * j * k;

        for(i=0; i<3; i++) {
            for(j=0; j<4; j++) {
                for(k=0; k<5; k++)
                    System.out.print(threeD[i][j][k] + " ");
                System.out.println();
            }
            System.out.println();
        }
    }
}
```

This program generates the following output:

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

```
0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12
```

```
0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24
```

Alternative Array Declaration Syntax

There is a second form that may be used to declare an array:

type[] var-name;

Here, the square brackets follow the type specifier, and not the name of the array variable. For example, the following two declarations are equivalent:

```
int a1[] = new int[3];
int[] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

```
int[] nums, nums2, nums3; // create three arrays
creates three array variables of type int. It is the same as writing
int nums[], nums2[], nums3[]; // create three arrays
```

The alternative declaration form is also useful when specifying an array as a return type for a method. Both forms are used in this book.

A Few Words About Strings

As you may have noticed, in the preceding discussion of data types and arrays there has been no mention of strings or a string data type. This is not because Java does not support such a type—it does. It is just that Java's string type, called **String**, is not a primitive type. Nor is it simply an array of characters. Rather, **String** defines an object, and a full description of it requires an understanding of several object-related features. As such, it will be covered later in this book, after objects are described. However, so that you can use simple strings in example programs, the following brief introduction is in order.

The **String** type is used to declare string variables. You can also declare arrays of strings. A quoted string constant can be assigned to a **String** variable. A variable of type **String** can

be assigned to another variable of type **String**. You can use an object of type **String** as an argument to **println()**. For example, consider the following fragment:

```
String str = "this is a test";
System.out.println(str);
```

Here, **str** is an object of type **String**. It is assigned the string "this is a test". This string is displayed by the **println()** statement.

As you will see later, **String** objects have many special features and attributes that make them quite powerful and easy to use. However, for the next few chapters, you will be using them only in their simplest form.

A Note to C/C++ Programmers About Pointers

If you are an experienced C/C++ programmer, then you know that these languages provide support for pointers. However, no mention of pointers has been made in this chapter. The reason for this is simple: Java does not support or allow pointers. (Or more properly, Java does not support pointers that can be accessed and/or modified by the programmer.) Java cannot allow pointers, because doing so would allow Java programs to breach the firewall between the Java execution environment and the host computer. (Remember, a pointer can be given any address in memory—even addresses that might be outside the Java run-time system.) Since C/C++ make extensive use of pointers, you might be thinking that their loss is a significant disadvantage to Java. However, this is not true. Java is designed in such a way that as long as you stay within the confines of the execution environment, you will never need to use a pointer, nor would there be any benefit in using one.

This page has been intentionally left blank

CHAPTER

4

Operators

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations. This chapter describes all of Java's operators except for the type comparison operator `instanceof`, which is examined in Chapter 13 and the new arrow operator (`->`), which is described in Chapter 15.

Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Result
<code>+</code>	Addition (also unary plus)
<code>-</code>	Subtraction (also unary minus)
<code>*</code>	Multiplication
<code>/</code>	Division
<code>%</code>	Modulus
<code>++</code>	Increment
<code>+=</code>	Addition assignment
<code>-=</code>	Subtraction assignment
<code>*=</code>	Multiplication assignment
<code>/=</code>	Division assignment
<code>%=</code>	Modulus assignment
<code>--</code>	Decrement

The operands of the arithmetic operators must be of a numeric type. You cannot use them on `boolean` types, but you can use them on `char` types, since the `char` type in Java is, essentially, a subset of `int`.

The Basic Arithmetic Operators

The basic arithmetic operations—addition, subtraction, multiplication, and division—all behave as you would expect for all numeric types. The unary minus operator negates its single operand. The unary plus operator simply returns the value of its operand. Remember that when the division operator is applied to an integer type, there will be no fractional component attached to the result.

The following simple example program demonstrates the arithmetic operators. It also illustrates the difference between floating-point division and integer division.

```
// Demonstrate the basic arithmetic operators.
class BasicMath {
    public static void main(String args[]) {
        // arithmetic using integers
        System.out.println("Integer Arithmetic");
        int a = 1 + 1;
        int b = a * 3;
        int c = b / 4;
        int d = c - a;
        int e = -d;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
        System.out.println("e = " + e);

        // arithmetic using doubles
        System.out.println("\nFloating Point Arithmetic");
        double da = 1 + 1;
        double db = da * 3;
        double dc = db / 4;
        double dd = dc - a;
        double de = -dd;
        System.out.println("da = " + da);
        System.out.println("db = " + db);
        System.out.println("dc = " + dc);
        System.out.println("dd = " + dd);
        System.out.println("de = " + de);
    }
}
```

When you run this program, you will see the following output:

```
Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e = 1

Floating Point Arithmetic
da = 2.0
db = 6.0
```

```
dc = 1.5
dd = -0.5
de = 0.5
```

The Modulus Operator

The modulus operator, `%`, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types. The following example program demonstrates the `%`:

```
// Demonstrate the % operator.
class Modulus {
    public static void main(String args[]) {
        int x = 42;
        double y = 42.25;

        System.out.println("x mod 10 = " + x % 10);
        System.out.println("y mod 10 = " + y % 10);
    }
}
```

When you run this program, you will get the following output:

```
x mod 10 = 2
y mod 10 = 2.25
```

Arithmetic Compound Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation with an assignment. As you probably know, statements like the following are quite common in programming:

```
a = a + 4;
```

In Java, you can rewrite this statement as shown here:

```
a += 4;
```

This version uses the `+= compound assignment operator`. Both statements perform the same action: they increase the value of `a` by 4.

Here is another example,

```
a = a % 2;
```

which can be expressed as

```
a %= 2;
```

In this case, the `%=` obtains the remainder of `a / 2` and puts that result back into `a`.

There are compound assignment operators for all of the arithmetic, binary operators. Thus, any statement of the form

var = var op expression;

can be rewritten as

```
var op= expression;
```

The compound assignment operators provide two benefits. First, they save you a bit of typing, because they are “shorthand” for their equivalent long forms. Second, in some cases they are more efficient than are their equivalent long forms. For these reasons, you will often see the compound assignment operators used in professionally written Java programs.

Here is a sample program that shows several *op=* assignments in action:

```
// Demonstrate several assignment operators.
class OpEquals {
    public static void main(String args[]) {
        int a = 1;
        int b = 2;
        int c = 3;

        a += 5;
        b *= 4;
        c += a * b;
        c %= 6;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
    }
}
```

The output of this program is shown here:

```
a = 6
b = 8
c = 3
```

Increment and Decrement

The `++` and the `--` are Java’s increment and decrement operators. They were introduced in Chapter 2. Here they will be discussed in detail. As you will see, they have some special properties that make them quite interesting. Let’s begin by reviewing precisely what the increment and decrement operators do.

The increment operator increases its operand by one. The decrement operator decreases its operand by one. For example, this statement:

```
x = x + 1;
```

can be rewritten like this by use of the increment operator:

```
x++;
```

Similarly, this statement:

```
x = x - 1;
```

is equivalent to

```
x--;
```

These operators are unique in that they can appear both in *postfix* form, where they follow the operand as just shown, and *prefix* form, where they precede the operand. In the foregoing examples, there is no difference between the prefix and postfix forms. However, when the increment and/or decrement operators are part of a larger expression, then a subtle, yet powerful, difference between these two forms appears. In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression. In postfix form, the previous value is obtained for use in the expression, and then the operand is modified. For example:

```
x = 42;
y = ++x;
```

In this case, **y** is set to 43 as you would expect, because the increment occurs *before* **x** is assigned to **y**. Thus, the line **y = ++x;** is the equivalent of these two statements:

```
x = x + 1;
y = x;
```

However, when written like this,

```
x = 42;
y = x++;
```

the value of **x** is obtained before the increment operator is executed, so the value of **y** is 42. Of course, in both cases **x** is set to 43. Here, the line **y = x++;** is the equivalent of these two statements:

```
y = x;
x = x + 1;
```

The following program demonstrates the increment operator.

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

The output of this program follows:

```
a = 2
b = 3
c = 4
d = 1
```

The Bitwise Operators

Java defines several *bitwise operators* that can be applied to the integer types: **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands. They are summarized in the following table:

Operator	Result
<code>~</code>	Bitwise unary NOT
<code>&</code>	Bitwise AND
<code> </code>	Bitwise OR
<code>^</code>	Bitwise exclusive OR
<code>>></code>	Shift right
<code>>>></code>	Shift right zero fill
<code><<</code>	Shift left
<code>&=</code>	Bitwise AND assignment
<code> =</code>	Bitwise OR assignment
<code>^=</code>	Bitwise exclusive OR assignment
<code>>>=</code>	Shift right assignment
<code>>>>=</code>	Shift right zero fill assignment
<code><<=</code>	Shift left assignment

Since the bitwise operators manipulate the bits within an integer: it is important to understand what effects such manipulations may have on a value. Specifically, it is useful to know how Java stores integer values and how it represents negative numbers. So, before continuing, let's briefly review these two topics.

All of the integer types are represented by binary numbers of varying bit widths. For example, the **byte** value for 42 in binary is 00101010, where each position represents a power of two, starting with 2^0 at the rightmost bit. The next bit position to the left would be 2^1 , or 2, continuing toward the left with 2^2 , or 4, then 8, 16, 32, and so on. So 42 has 1 bits set at positions 1, 3, and 5 (counting from 0 at the right); thus, 42 is the sum of $2^1 + 2^3 + 2^5$, which is $2 + 8 + 32$.

All of the integer types (except **char**) are signed integers. This means that they can represent negative values as well as positive ones. Java uses an encoding known as *two's complement*, which means that negative numbers are represented by inverting (changing 1's to 0's and vice versa) all of the bits in a value, then adding 1 to the result. For example, -42 is represented by inverting all of the bits in 42, or 00101010, which yields 11010101, then adding 1, which results in 11010110, or -42. To decode a negative number, first invert all

of the bits, then add 1. For example, -42 , or 11010110 inverted, yields 00101001 , or 41 , so when you add 1 you get 42 .

The reason Java (and most other computer languages) uses two's complement is easy to see when you consider the issue of *zero crossing*. Assuming a **byte** value, zero is represented by 00000000 . In one's complement, simply inverting all of the bits creates 11111111 , which creates negative zero. The trouble is that negative zero is invalid in integer math. This problem is solved by using two's complement to represent negative values. When using two's complement, 1 is added to the complement, producing 100000000 . This produces a 1 bit too far to the left to fit back into the **byte** value, resulting in the desired behavior, where -0 is the same as 0 , and 11111111 is the encoding for -1 . Although we used a **byte** value in the preceding example, the same basic principle applies to all of Java's integer types.

Because Java uses two's complement to store negative numbers—and because all integers are signed values in Java—applying the bitwise operators can easily produce unexpected results. For example, turning on the high-order bit will cause the resulting value to be interpreted as a negative number, whether this is what you intended or not. To avoid unpleasant surprises, just remember that the high-order bit determines the sign of an integer no matter how that high-order bit gets set.

The Bitwise Logical Operators

The bitwise logical operators are **&**, **|**, **^**, and **~**. The following table shows the outcome of each operation. In the discussion that follows, keep in mind that the bitwise operators are applied to each individual bit within each operand.

A	B	A B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

The Bitwise NOT

Also called the *bitwise complement*, the unary NOT operator, **~**, inverts all of the bits of its operand. For example, the number 42 , which has the following bit pattern:

00101010

becomes

11010101

after the NOT operator is applied.

The Bitwise AND

The AND operator, **&**, produces a 1 bit if both operands are also 1. A zero is produced in all other cases. Here is an example:

00101010	42
&00001111	15
<hr/>	
00001010	10

00001010 10

The Bitwise OR

The OR operator, `|`, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

00101010	42
00001111	15
<hr/>	
00101111	47

The Bitwise XOR

The XOR operator, `^`, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero. The following example shows the effect of the `^`. This example also demonstrates a useful attribute of the XOR operation. Notice how the bit pattern of 42 is inverted wherever the second operand has a 1 bit. Wherever the second operand has a 0 bit, the first operand is unchanged. You will find this property useful when performing some types of bit manipulations.

00101010	42
^ 00001111	15
<hr/>	
00100101	37

Using the Bitwise Logical Operators

The following program demonstrates the bitwise logical operators:

```
// Demonstrate the bitwise logical operators.
class BitLogic {
    public static void main(String args[]) {
        String binary[] = {
            "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
            "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
        };
        int a = 3; // 0 + 2 + 1 or 0011 in binary
        int b = 6; // 4 + 2 + 0 or 0110 in binary
        int c = a | b;
        int d = a & b;
        int e = a ^ b;
        int f = (~a & b) | (a & ~b);
        int g = ~a & 0x0f;

        System.out.println("      a = " + binary[a]);
        System.out.println("      b = " + binary[b]);
        System.out.println(" a|b = " + binary[c]);
        System.out.println(" a&b = " + binary[d]);
        System.out.println(" a^b = " + binary[e]);
        System.out.println(" ~a&b|a&~b = " + binary[f]);
        System.out.println("      ~a = " + binary[g]);
    }
}
```

In this example, **a** and **b** have bit patterns that present all four possibilities for two binary digits: 0-0, 0-1, 1-0, and 1-1. You can see how the `|` and `&` operate on each bit by the

results in **c** and **d**. The values assigned to **e** and **f** are the same and illustrate how the \wedge works. The string array named **binary** holds the human-readable, binary representation of the numbers 0 through 15. In this example, the array is indexed to show the binary representation of each result. The array is constructed such that the correct string representation of a binary value **n** is stored in **binary[n]**. The value of $\sim a$ is ANDed with **0x0f** (0000 1111 in binary) in order to reduce its value to less than 16, so it can be printed by use of the **binary** array. Here is the output from this program:

```
a = 0011
b = 0110
a|b = 0111
a&b = 0010
a^b = 0101
~a&b|a&~b = 0101
~a = 1100
```

The Left Shift

The left shift operator, `<<`, shifts all of the bits in a value to the left a specified number of times. It has this general form:

value `<<` *num*

Here, *num* specifies the number of positions to left-shift the value in *value*. That is, the `<<` moves all of the bits in the specified value to the left by the number of bit positions specified by *num*. For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right. This means that when a left shift is applied to an **int** operand, bits are lost once they are shifted past bit position 31. If the operand is a **long**, then bits are lost after bit position 63.

Java's automatic type promotions produce unexpected results when you are shifting **byte** and **short** values. As you know, **byte** and **short** values are promoted to **int** when an expression is evaluated. Furthermore, the result of such an expression is also an **int**. This means that the outcome of a left shift on a **byte** or **short** value will be an **int**, and the bits shifted left will not be lost until they shift past bit position 31. Furthermore, a negative **byte** or **short** value will be sign-extended when it is promoted to **int**. Thus, the high-order bits will be filled with 1's. For these reasons, to perform a left shift on a **byte** or **short** implies that you must discard the high-order bytes of the **int** result. For example, if you left-shift a **byte** value, that value will first be promoted to **int** and then shifted. This means that you must discard the top three bytes of the result if what you want is the result of a shifted **byte** value. The easiest way to do this is to simply cast the result back into a **byte**. The following program demonstrates this concept:

```
// Left shifting a byte value.
class ByteShift {
    public static void main(String args[]) {
        byte a = 64, b;
        int i;

        i = a << 2;
        b = (byte) (a << 2);
```

```

        System.out.println("Original value of a: " + a);
        System.out.println("i and b: " + i + " " + b);
    }
}

```

The output generated by this program is shown here:

```

Original value of a: 64
i and b: 256 0

```

Since **a** is promoted to **int** for the purposes of evaluation, left-shifting the value 64 (0100 0000) twice results in **i** containing the value 256 (1 0000 0000). However, the value in **b** contains 0 because after the shift, the low-order byte is now zero. Its only 1 bit has been shifted out.

Since each left shift has the effect of doubling the original value, programmers frequently use this fact as an efficient alternative to multiplying by 2. But you need to watch out. If you shift a 1 bit into the high-order position (bit 31 or 63), the value will become negative. The following program illustrates this point:

```

// Left shifting as a quick way to multiply by 2.
class MultByTwo {
    public static void main(String args[]) {
        int i;
        int num = 0xFFFFFE;

        for(i=0; i<4; i++) {
            num = num << 1;
            System.out.println(num);
        }
    }
}

```

The program generates the following output:

```

536870908
1073741816
2147483632
-32

```

The starting value was carefully chosen so that after being shifted left 4 bit positions, it would produce -32. As you can see, when a 1 bit is shifted into bit 31, the number is interpreted as negative.

The Right Shift

The right shift operator, **>>**, shifts all of the bits in a value to the right a specified number of times. Its general form is shown here:

value >> num

Here, *num* specifies the number of positions to right-shift the value in *value*. That is, the **>>** moves all of the bits in the specified value to the right the number of bit positions specified by *num*.

The following code fragment shifts the value 32 to the right by two positions, resulting in **a** being set to 8:

```
int a = 32;
a = a >> 2; // a now contains 8
```

When a value has bits that are “shifted off,” those bits are lost. For example, the next code fragment shifts the value 35 to the right two positions, which causes the two low-order bits to be lost, resulting again in **a** being set to 8:

```
int a = 35;
a = a >> 2; // a contains 8
```

Looking at the same operation in binary shows more clearly how this happens:

```
00100011 35
>> 2
00001000 8
```

Each time you shift a value to the right, it divides that value by two—and discards any remainder. In some cases, you can take advantage of this for high-performance integer division by 2.

When you are shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit. This is called *sign extension* and serves to preserve the sign of negative numbers when you shift them right. For example, $-8 \gg 1$ is -4 , which, in binary, is

```
11111000 -8
>> 1
11111100 -4
```

It is interesting to note that if you shift -1 right, the result always remains -1 , since sign extension keeps bringing in more ones in the high-order bits.

Sometimes it is not desirable to sign-extend values when you are shifting them to the right. For example, the following program converts a **byte** value to its hexadecimal string representation. Notice that the shifted value is masked by ANDing it with **0x0f** to discard any sign-extended bits so that the value can be used as an index into the array of hexadecimal characters.

```
// Masking sign extension.
class HexByte {
    static public void main(String args[]) {
        char hex[] = {
            '0', '1', '2', '3', '4', '5', '6', '7',
            '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
        };

        byte b = (byte) 0xf1;

        System.out.println("b = 0x" + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
    }
}
```

Here is the output of this program:

```
b = 0xf1
```

The Unsigned Right Shift

As you have just seen, the `>>` operator automatically fills the high-order bit with its previous contents each time a shift occurs. This preserves the sign of the value. However, sometimes this is undesirable. For example, if you are shifting something that does not represent a numeric value, you may not want sign extension to take place. This situation is common when you are working with pixel-based values and graphics. In these cases, you will generally want to shift a zero into the high-order bit no matter what its initial value was. This is known as an *unsigned shift*. To accomplish this, you will use Java's unsigned, shift-right operator, `>>>`, which always shifts zeros into the high-order bit.

The following code fragment demonstrates the `>>>`. Here, `a` is set to `-1`, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets `a` to 255.

```
int a = -1;
a = a >>> 24;
```

Here is the same operation in binary form to further illustrate what is happening:

```
11111111 11111111 11111111 11111111 -1 in binary as an int
>>>24
00000000 00000000 00000000 11111111 255 in binary as an int
```

The `>>>` operator is often not as useful as you might like, since it is only meaningful for 32- and 64-bit values. Remember, smaller values are automatically promoted to `int` in expressions. This means that sign-extension occurs and that the shift will take place on a 32-bit rather than on an 8- or 16-bit value. That is, one might expect an unsigned right shift on a `byte` value to zero-fill beginning at bit 7. But this is not the case, since it is a 32-bit value that is actually being shifted. The following program demonstrates this effect:

```
// Unsigned shifting a byte value.
class ByteUShift {
    static public void main(String args[]) {
        char hex[] = {
            '0', '1', '2', '3', '4', '5', '6', '7',
            '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
        };
        byte b = (byte) 0xf1;
        byte c = (byte) (b >> 4);
        byte d = (byte) (b >>> 4);
        byte e = (byte) ((b & 0xff) >> 4);

        System.out.println("          b = 0x"
            + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
        System.out.println("          b >> 4 = 0x"
            + hex[(c >> 4) & 0x0f] + hex[c & 0x0f]);
        System.out.println("          b >>> 4 = 0x"
            + hex[(d >> 4) & 0x0f] + hex[d & 0x0f]);
```

```

        System.out.println((b & 0xff) >> 4 = 0x"
            + hex[(e >> 4) & 0x0f] + hex[e & 0x0f]);
    }
}

```

The following output of this program shows how the `>>>` operator appears to do nothing when dealing with bytes. The variable **b** is set to an arbitrary negative **byte** value for this demonstration. Then **c** is assigned the **byte** value of **b** shifted right by four, which is 0xff because of the expected sign extension. Then **d** is assigned the **byte** value of **b** unsigned shifted right by four, which you might have expected to be 0x0f, but is actually 0xff because of the sign extension that happened when **b** was promoted to **int** before the shift. The last expression sets **e** to the **byte** value of **b** masked to 8 bits using the AND operator, then shifted right by four, which produces the expected value of 0x0f. Notice that the unsigned shift right operator was not used for **d**, since the state of the sign bit after the AND was known.

```

b = 0xf1
b >> 4 = 0xff
b >>> 4 = 0xff
(b & 0xff) >> 4 = 0x0f

```

Bitwise Operator Compound Assignments

All of the binary bitwise operators have a compound form similar to that of the algebraic operators, which combines the assignment with the bitwise operation. For example, the following two statements, which shift the value in **a** right by four bits, are equivalent:

```

a = a >> 4;
a >>= 4;

```

Likewise, the following two statements, which result in **a** being assigned the bitwise expression **a OR b**, are equivalent:

```

a = a | b;
a |= b;

```

The following program creates a few integer variables and then uses compound bitwise operator assignments to manipulate the variables:

```

class OpBitEquals {
    public static void main(String args[]) {
        int a = 1;
        int b = 2;
        int c = 3;

        a |= 4;
        b >>= 1;
        c <<= 1;
        a ^= c;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
    }
}

```

The output of this program is shown here:

```
a = 3
b = 1
c = 6
```

Relational Operators

The *relational operators* determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements.

Any type in Java, including integers, floating-point numbers, characters, and Booleans can be compared using the equality test, **==**, and the inequality test, **!=**. Notice that in Java equality is denoted with two equal signs, not one. (Remember: a single equal sign is the assignment operator.) Only numeric types can be compared using the ordering operators. That is, only integer, floating-point, and character operands may be compared to see which is greater or less than the other.

As stated, the result produced by a relational operator is a **boolean** value. For example, the following code fragment is perfectly valid:

```
int a = 4;
int b = 1;
boolean c = a < b;
```

In this case, the result of **a<b** (which is **false**) is stored in **c**.

If you are coming from a C/C++ background, please note the following. In C/C++, these types of statements are very common:

```
int done;
//...
if(!done)... // Valid in C/C++
if(done)... // but not in Java.
```

In Java, these statements must be written like this:

```
if(done == 0)... // This is Java-style.
if(done != 0)...
```

The reason is that Java does not define true and false in the same way as C/C++. In C/C++, true is any nonzero value and false is zero. In Java, **true** and **false** are nonnumeric values that do not relate to zero or nonzero. Therefore, to test for zero or nonzero, you must explicitly employ one or more of the relational operators.

Boolean Logical Operators

The Boolean logical operators shown here operate only on **boolean** operands. All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value.

Operator	Result
&	Logical AND
	Logical OR
^	Logical XOR (exclusive OR)
	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

The logical Boolean operators, **&**, **|**, and **^**, operate on **boolean** values in the same way that they operate on the bits of an integer. The logical **!** operator inverts the Boolean state: **!true == false** and **!false == true**. The following table shows the effect of each logical operation:

A	B	A B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Here is a program that is almost the same as the **BitLogic** example shown earlier, but it operates on **boolean** logical values instead of binary bits:

```
// Demonstrate the boolean logical operators.
class BoolLogic {
    public static void main(String args[]) {
        boolean a = true;
        boolean b = false;
        boolean c = a | b;
        boolean d = a & b;
```

```

        boolean e = a ^ b;
        boolean f = (!a & b) | (a & !b);
        boolean g = !a;
        System.out.println("      a = " + a);
        System.out.println("      b = " + b);
        System.out.println("      a|b = " + c);
        System.out.println("      a&b = " + d);
        System.out.println("      a^b = " + e);
        System.out.println("!a&b|a&!b = " + f);
        System.out.println("      !a = " + g);
    }
}

```

After running this program, you will see that the same logical rules apply to **boolean** values as they did to bits. As you can see from the following output, the string representation of a Java **boolean** value is one of the literal values **true** or **false**:

```

a = true
b = false
a|b = true
a&b = false
a^b = true
!a&b|a&!b = true
!a = false

```

Short-Circuit Logical Operators

Java provides two interesting Boolean operators not found in some other computer languages. These are secondary versions of the Boolean AND and OR operators, and are commonly known as *short-circuit* logical operators. As you can see from the preceding table, the OR operator results in **true** when **A** is **true**, no matter what **B** is. Similarly, the AND operator results in **false** when **A** is **false**, no matter what **B** is. If you use the **||** and **&&** forms, rather than the **|** and **&** forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone. This is very useful when the right-hand operand depends on the value of the left one in order to function properly. For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

```
if (denom != 0 && num / denom > 10)
```

Since the short-circuit form of AND (**&&**) is used, there is no risk of causing a run-time exception when **denom** is zero. If this line of code were written using the single **&** version of AND, both sides would be evaluated, causing a run-time exception when **denom** is zero.

It is standard practice to use the short-circuit forms of AND and OR in cases involving Boolean logic, leaving the single-character versions exclusively for bitwise operations. However, there are exceptions to this rule. For example, consider the following statement:

```
if(c==1 & e++ < 100) d = 100;
```

Here, using a single **&** ensures that the increment operation will be applied to **e** whether **c** is equal to 1 or not.

NOTE The formal specification for Java refers to the short-circuit operators as the *conditional-and* and the *conditional-or*.

The Assignment Operator

You have been using the assignment operator since Chapter 2. Now it is time to take a formal look at it. The *assignment operator* is the single equal sign, `=`. The assignment operator works in Java much as it does in any other computer language. It has this general form:

`var = expression;`

Here, the type of `var` must be compatible with the type of `expression`.

The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables `x`, `y`, and `z` to 100 using a single statement. This works because the `=` is an operator that yields the value of the right-hand expression. Thus, the value of `z = 100` is 100, which is then assigned to `y`, which in turn is assigned to `x`. Using a “chain of assignment” is an easy way to set a group of variables to a common value.

The ? Operator

Java includes a special *ternary* (three-way) *operator* that can replace certain types of if-then-else statements. This operator is the `?`. It can seem somewhat confusing at first, but the `?` can be used very effectively once mastered. The `?` has this general form:

`expression1 ? expression2 : expression3`

Here, `expression1` can be any expression that evaluates to a **boolean** value. If `expression1` is **true**, then `expression2` is evaluated; otherwise, `expression3` is evaluated. The result of the `?` operation is that of the expression evaluated. Both `expression2` and `expression3` are required to return the same (or compatible) type, which can't be **void**.

Here is an example of the way that the `?` is employed:

```
ratio = denom == 0 ? 0 : num / denom;
```

When Java evaluates this assignment expression, it first looks at the expression to the *left* of the question mark. If `denom` equals zero, then the expression *between* the question mark and the colon is evaluated and used as the value of the entire `?` expression. If `denom` does not equal zero, then the expression *after* the colon is evaluated and used for the value of the entire `?` expression. The result produced by the `?` operator is then assigned to `ratio`.

Here is a program that demonstrates the `?` operator. It uses it to obtain the absolute value of a variable.

```
// Demonstrate ?.
class Ternary {
    public static void main(String args[]) {
        int i, k;
```

```

i = 10;
k = i < 0 ? -i : i; // get absolute value of i
System.out.print("Absolute value of ");
System.out.println(i + " is " + k);

i = -10;
k = i < 0 ? -i : i; // get absolute value of i
System.out.print("Absolute value of ");
System.out.println(i + " is " + k);
}
}

```

The output generated by the program is shown here:

```

Absolute value of 10 is 10
Absolute value of -10 is 10

```

Operator Precedence

Table 4-1 shows the order of precedence for Java operators, from highest to lowest. Operators in the same row are equal in precedence. In binary operations, the order of evaluation is left to right (except for assignment, which evaluates right to left). Although they are technically separators, the [], (), and . can also act like operators. In that capacity, they would have the highest precedence. Also, notice the arrow operator (->). It was added by JDK 8 and is used in lambda expressions.

Highest						
++ (postfix)	-- (postfix)					
++ (prefix)	-- (prefix)	~	!	+ (unary)	- (unary)	(<i>type-cast</i>)
*	/	%				
+	-					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
&&						
?:						
->						
=	op=					
Lowest						

Table 4-1 The Precedence of the Java Operators

Using Parentheses

Parentheses raise the precedence of the operations that are inside them. This is often necessary to obtain the result you desire. For example, consider the following expression:

```
a >> b + 3
```

This expression first adds 3 to **b** and then shifts **a** right by that result. That is, this expression can be rewritten using redundant parentheses like this:

```
a >> (b + 3)
```

However, if you want to first shift **a** right by **b** positions and then add 3 to that result, you will need to parenthesize the expression like this:

```
(a >> b) + 3
```

In addition to altering the normal precedence of an operator, parentheses can sometimes be used to help clarify the meaning of an expression. For anyone reading your code, a complicated expression can be difficult to understand. Adding redundant but clarifying parentheses to complex expressions can help prevent confusion later. For example, which of the following expressions is easier to read?

```
a | 4 + c >> b & 7  
(a | (((4 + c) >> b) & 7))
```

One other point: parentheses (redundant or not) do not degrade the performance of your program. Therefore, adding parentheses to reduce ambiguity does not negatively affect your program.

This page has been intentionally left blank

CHAPTER

5

Control Statements

A programming language uses *control* statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: selection, iteration, and jump. *Selection* statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. *Iteration* statements enable program execution to repeat one or more statements (that is, iteration statements form loops). *Jump* statements allow your program to execute in a nonlinear fashion. All of Java's control statements are examined here.

Java's Selection Statements

Java supports two selection statements: **if** and **switch**. These statements allow you to control the flow of your program's execution based upon conditions known only during run time. You will be pleasantly surprised by the power and flexibility contained in these two statements.

if

The **if** statement was introduced in Chapter 2. It is examined in detail here. The **if** statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the **if** statement:

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

Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a *block*). The *condition* is any expression that returns a **boolean** value. The **else** clause is optional.

The **if** works like this: If the *condition* is true, then *statement1* is executed. Otherwise, *statement2* (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

```
int a, b;  
//...  
if(a < b) a = 0;  
else b = 0;
```

Here, if **a** is less than **b**, then **a** is set to zero. Otherwise, **b** is set to zero. In no case are they both set to zero.

Most often, the expression used to control the **if** will involve the relational operators. However, this is not technically necessary. It is possible to control the **if** using a single **boolean** variable, as shown in this code fragment:

```
boolean dataAvailable;  
//...  
if (dataAvailable)  
    ProcessData();  
else  
    waitForMoreData();
```

Remember, only one statement can appear directly after the **if** or the **else**. If you want to include more statements, you'll need to create a block, as in this fragment:

```
int bytesAvailable;  
// ...  
if (bytesAvailable > 0) {  
    ProcessData();  
    bytesAvailable -= n;  
} else  
    waitForMoreData();
```

Here, both statements within the **if** block will execute if **bytesAvailable** is greater than zero.

Some programmers find it convenient to include the curly braces when using the **if**, even when there is only one statement in each clause. This makes it easy to add another statement at a later date, and you don't have to worry about forgetting the braces. In fact, forgetting to define a block when one is needed is a common cause of errors. For example, consider the following code fragment:

```
int bytesAvailable;  
// ...  
if (bytesAvailable > 0) {  
    ProcessData();  
    bytesAvailable -= n;  
} else  
    waitForMoreData();  
bytesAvailable = n;
```

It seems clear that the statement **bytesAvailable = n;** was intended to be executed inside the **else** clause, because of the indentation level. However, as you recall, whitespace is insignificant to Java, and there is no way for the compiler to know what was intended. This code will compile without complaint, but it will behave incorrectly when run. The preceding example is fixed in the code that follows:

```
int bytesAvailable;  
// ...  
if (bytesAvailable > 0) {  
    ProcessData();  
    bytesAvailable -= n;  
} else {
```

```

waitForMoreData();
bytesAvailable = n;
}

```

Nested ifs

A *nested if* is an **if** statement that is the target of another **if** or **else**. Nested **ifs** are very common in programming. When you nest **ifs**, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**. Here is an example:

```

if(i == 10) {
    if(j < 20) a = b;
    if(k > 100) c = d; // this if is
        else a = c;      // associated with this else
    }
else a = d;           // this else refers to if(i == 10)

```

As the comments indicate, the final **else** is not associated with **if(j<20)** because it is not in the same block (even though it is the nearest **if** without an **else**). Rather, the final **else** is associated with **if(i==10)**. The inner **else** refers to **if(k>100)** because it is the closest **if** within the same block.

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested **ifs** is the *if-else-if ladder*. It looks like this:

```

if(condition)
    statement;
else if(condition)
    statement;
else if(condition)
    statement;
.
.
.
else
    statement;

```

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final **else** statement will be executed. The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed. If there is no final **else** and all other conditions are **false**, then no action will take place.

Here is a program that uses an **if-else-if** ladder to determine which season a particular month is in.

```

// Demonstrate if-else-if statements.
class IfElse {
    public static void main(String args[]) {
        int month = 4; // April
        String season;

```

```

        if(month == 12 || month == 1 || month == 2)
            season = "Winter";
        else if(month == 3 || month == 4 || month == 5)
            season = "Spring";
        else if(month == 6 || month == 7 || month == 8)
            season = "Summer";
        else if(month == 9 || month == 10 || month == 11)
            season = "Autumn";
        else
            season = "Bogus Month";

        System.out.println("April is in the " + season + ".");
    }
}

```

Here is the output produced by the program:

```
April is in the Spring.
```

You might want to experiment with this program before moving on. As you will find, no matter what value you give **month**, one and only one assignment statement within the ladder will be executed.

switch

The **switch** statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of **if-else-if** statements. Here is the general form of a **switch** statement:

```

switch (expression) {
    case value1:
        // statement sequence
        break;
    case value2:
        // statement sequence
        break;
    .
    .
    .
    case valueN:
        // statement sequence
        break;
    default:
        // default statement sequence
}

```

For versions of Java prior to JDK 7, *expression* must be of type **byte**, **short**, **int**, **char**, or an enumeration. (Enumerations are described in Chapter 12.) Beginning with JDK 7, *expression*

can also be of type **String**. Each value specified in the **case** statements must be a unique constant expression (such as a literal value). Duplicate **case** values are not allowed. The type of each value must be compatible with the type of *expression*.

The **switch** statement works like this: The value of the expression is compared with each of the values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed. If none of the constants matches the value of the expression, then the **default** statement is executed. However, the **default** statement is optional. If no **case** matches and no **default** is present, then no further action is taken.

The **break** statement is used inside the **switch** to terminate a statement sequence. When a **break** statement is encountered, execution branches to the first line of code that follows the entire **switch** statement. This has the effect of “jumping out” of the **switch**.

Here is a simple example that uses a **switch** statement:

```
// A simple example of the switch.
class SampleSwitch {
    public static void main(String args[]) {
        for(int i=0; i<6; i++)
            switch(i) {
                case 0:
                    System.out.println("i is zero.");
                    break;
                case 1:
                    System.out.println("i is one.");
                    break;
                case 2:
                    System.out.println("i is two.");
                    break;
                case 3:
                    System.out.println("i is three.");
                    break;
                default:
                    System.out.println("i is greater than 3.");
            }
    }
}
```

The output produced by this program is shown here:

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

As you can see, each time through the loop, the statements associated with the **case** constant that matches **i** are executed. All others are bypassed. After **i** is greater than 3, no **case** statements match, so the **default** statement is executed.

The **break** statement is optional. If you omit the **break**, execution will continue on into the next **case**. It is sometimes desirable to have multiple **cases** without **break** statements between them. For example, consider the following program:

```
// In a switch, break statements are optional.
class MissingBreak {
    public static void main(String args[]) {
        for(int i=0; i<12; i++)
            switch(i) {
                case 0:
                case 1:
                case 2:
                case 3:
                case 4:
                    System.out.println("i is less than 5");
                    break;
                case 5:
                case 6:
                case 7:
                case 8:
                case 9:
                    System.out.println("i is less than 10");
                    break;
                default:
                    System.out.println("i is 10 or more");
            }
    }
}
```

This program generates the following output:

```
i is less than 5
i is less than 10
i is 10 or more
i is 10 or more
```

As you can see, execution falls through each **case** until a **break** statement (or the end of the **switch**) is reached.

While the preceding example is, of course, contrived for the sake of illustration, omitting the **break** statement has many practical applications in real programs. To sample its more realistic usage, consider the following rewrite of the season example shown earlier. This version uses a **switch** to provide a more efficient implementation.

```
// An improved version of the season program.
class Switch {
    public static void main(String args[]) {
        int month = 4;
```

```
String season;

switch (month) {
    case 12:
    case 1:
    case 2:
        season = "Winter";
        break;
    case 3:
    case 4:
    case 5:
        season = "Spring";
        break;
    case 6:
    case 7:
    case 8:
        season = "Summer";
        break;
    case 9:
    case 10:
    case 11:
        season = "Autumn";
        break;
    default:
        season = "Bogus Month";
}
System.out.println("April is in the " + season + ".");
}
```

As mentioned, beginning with JDK 7, you can use a string to control a **switch** statement. For example,

```
// Use a string to control a switch statement.

class StringSwitch {
    public static void main(String args[]) {

        String str = "two";

        switch(str) {
            case "one":
                System.out.println("one");
                break;
            case "two":
                System.out.println("two");
                break;
            case "three":
                System.out.println("three");
                break;
            default:
                System.out.println("no match");
                break;
        }
    }
}
```

As you would expect, the output from the program is

```
two
```

The string contained in **str** (which is "two" in this program) is tested against the **case** constants. When a match is found (as it is in the second **case**), the code sequence associated with that sequence is executed.

Being able to use strings in a **switch** statement streamlines many situations. For example, using a string-based **switch** is an improvement over using the equivalent sequence of **if/else** statements. However, switching on strings can be more expensive than switching on integers. Therefore, it is best to switch on strings only in cases in which the controlling data is already in string form. In other words, don't use strings in a **switch** unnecessarily.

Nested switch Statements

You can use a **switch** as part of the statement sequence of an outer **switch**. This is called a *nested switch*. Since a **switch** statement defines its own block, no conflicts arise between the **case** constants in the inner **switch** and those in the outer **switch**. For example, the following fragment is perfectly valid:

```
switch(count) {
    case 1:
        switch(target) { // nested switch
            case 0:
                System.out.println("target is zero");
                break;
            case 1: // no conflicts with outer switch
                System.out.println("target is one");
                break;
        }
        break;
    case 2: // ...
}
```

Here, the **case 1:** statement in the inner **switch** does not conflict with the **case 1:** statement in the outer **switch**. The **count** variable is compared only with the list of cases at the outer level. If **count** is 1, then **target** is compared with the inner list cases.

In summary, there are three important features of the **switch** statement to note:

- The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of Boolean expression. That is, the **switch** looks only for a match between the value of the expression and one of its **case** constants.
- No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement and an enclosing outer **switch** can have **case** constants in common.
- A **switch** statement is usually more efficient than a set of nested **ifs**.

The last point is particularly interesting because it gives insight into how the Java compiler works. When it compiles a **switch** statement, the Java compiler will inspect each of the **case** constants and create a "jump table" that it will use for selecting the path of execution depending on the value of the expression. Therefore, if you need to select among a large

group of values, a **switch** statement will run much faster than the equivalent logic coded using a sequence of **if-elses**. The compiler can do this because it knows that the **case** constants are all the same type and simply must be compared for equality with the **switch** expression. The compiler has no such knowledge of a long list of **if** expressions.

Iteration Statements

Java's iteration statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops*. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. As you will see, Java has a loop to fit any programming need.

while

The **while** loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
while(condition) {  
    // body of loop  
}
```

The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

Here is a **while** loop that counts down from 10, printing exactly ten lines of "tick":

```
// Demonstrate the while loop.  
class While {  
    public static void main(String args[]) {  
        int n = 10;  
  
        while(n > 0) {  
            System.out.println("tick " + n);  
            n--;  
        }  
    }  
}
```

When you run this program, it will "tick" ten times:

```
tick 10  
tick 9  
tick 8  
tick 7  
tick 6  
tick 5  
tick 4  
tick 3  
tick 2  
tick 1
```

Since the **while** loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with. For example, in the following fragment, the call to **println()** is never executed:

```
int a = 10, b = 20;  
  
while(a > b)  
    System.out.println("This will not be displayed");
```

The body of the **while** (or any other of Java's loops) can be empty. This is because a *null statement* (one that consists only of a semicolon) is syntactically valid in Java. For example, consider the following program:

```
// The target of a loop can be empty.  
class NoBody {  
    public static void main(String args[]) {  
        int i, j;  
  
        i = 100;  
        j = 200;  
  
        // find midpoint between i and j  
        while(++i < --j); // no body in this loop  
  
        System.out.println("Midpoint is " + i);  
    }  
}
```

This program finds the midpoint between **i** and **j**. It generates the following output:

```
Midpoint is 150
```

Here is how this **while** loop works. The value of **i** is incremented, and the value of **j** is decremented. These values are then compared with one another. If the new value of **i** is still less than the new value of **j**, then the loop repeats. If **i** is equal to or greater than **j**, the loop stops. Upon exit from the loop, **i** will hold a value that is midway between the original values of **i** and **j**. (Of course, this procedure only works when **i** is less than **j** to begin with.) As you can see, there is no need for a loop body; all of the action occurs within the conditional expression, itself. In professionally written Java code, short loops are frequently coded without bodies when the controlling expression can handle all of the details itself.

do-while

As you just saw, if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that: the **do-while**. The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

```
do {
    // body of loop
} while (condition);
```

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, *condition* must be a Boolean expression.

Here is a reworked version of the “tick” program that demonstrates the **do-while** loop. It generates the same output as before.

```
// Demonstrate the do-while loop.
class DoWhile {
    public static void main(String args[]) {
        int n = 10;

        do {
            System.out.println("tick " + n);
            n--;
        } while(n > 0);
    }
}
```

The loop in the preceding program, while technically correct, can be written more efficiently as follows:

```
do {
    System.out.println("tick " + n);
} while(--n > 0);
```

In this example, the expression **(--n > 0)** combines the decrement of **n** and the test for zero into one expression. Here is how it works. First, the **--n** statement executes, decrementing **n** and returning the new value of **n**. This value is then compared with zero. If it is greater than zero, the loop continues; otherwise, it terminates.

The **do-while** loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once. Consider the following program, which implements a very simple help system for Java's selection and iteration statements:

```
// Using a do-while to process a menu selection
class Menu {
    public static void main(String args[])
        throws java.io.IOException {
        char choice;

        do {
            System.out.println("Help on: ");
            System.out.println(" 1. if");
            System.out.println(" 2. switch");
            System.out.println(" 3. while");
            System.out.println(" 4. do-while");
            System.out.println(" 5. for\n");
            System.out.println("Choose one:");

    
```

```

choice = (char) System.in.read();
} while( choice < '1' || choice > '5');

System.out.println("\n");

switch(choice) {
    case '1':
        System.out.println("The if:\n");
        System.out.println("if(condition) statement;");
        System.out.println("else statement;");
        break;
    case '2':
        System.out.println("The switch:\n");
        System.out.println("switch(expression) { ");
        System.out.println("    case constant:");
        System.out.println("        statement sequence");
        System.out.println("        break; ");
        System.out.println("    //... ");
        System.out.println("} ");
        break;
    case '3':
        System.out.println("The while:\n");
        System.out.println("while(condition) statement;");
        break;
    case '4':
        System.out.println("The do-while:\n");
        System.out.println("do { ");
        System.out.println("    statement; ");
        System.out.println("} while (condition); ");
        break;
    case '5':
        System.out.println("The for:\n");
        System.out.println("for(init; condition; iteration) ");
        System.out.println("    statement; ");
        break;
}
}
}

```

Here is a sample run produced by this program:

```
Help on:  
    1. if  
    2. switch  
    3. while  
    4. do-while  
    5. for  
Choose one:  
4  
The do-while:  
do {  
    statement;  
} while (condition);
```

In the program, the **do-while** loop is used to verify that the user has entered a valid choice. If not, then the user is reprompted. Since the menu must be displayed at least once, the **do-while** is the perfect loop to accomplish this.

A few other points about this example: Notice that characters are read from the keyboard by calling **System.in.read()**. This is one of Java's console input functions. Although Java's console I/O methods won't be discussed in detail until Chapter 13, **System.in.read()** is used here to obtain the user's choice. It reads characters from standard input (returned as integers, which is why the return value was cast to **char**). By default, standard input is line buffered, so you must press ENTER before any characters that you type will be sent to your program.

Java's console input can be a bit awkward to work with. Further, most real-world Java programs will be graphical and window-based. For these reasons, not much use of console input has been made in this book. However, it is useful in this context. One other point to consider: Because **System.in.read()** is being used, the program must specify the **throws java.io.IOException** clause. This line is necessary to handle input errors. It is part of Java's exception handling features, which are discussed in Chapter 10.

for

You were introduced to a simple form of the **for** loop in Chapter 2. As you will see, it is a powerful and versatile construct.

Beginning with JDK 5, there are two forms of the **for** loop. The first is the traditional form that has been in use since the original version of Java. The second is the newer “for-each” form. Both types of **for** loops are discussed here, beginning with the traditional form.

Here is the general form of the traditional **for** statement:

```
for(initialization; condition; iteration) {
    // body
}
```

If only one statement is being repeated, there is no need for the curly braces.

The **for** loop operates as follows. When the loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loop control variable*, which acts as a counter that controls the loop. It is important to understand that the initialization expression is executed only once. Next, *condition* is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the *iteration* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

Here is a version of the “tick” program that uses a **for** loop:

```
// Demonstrate the for loop.
class ForTick {
    public static void main(String args[]) {
        int n;

        for(n=10; n>0; n--)
    }
}
```

```

        System.out.println("tick " + n);
    }
}

```

Declaring Loop Control Variables Inside the for Loop

Often the variable that controls a **for** loop is needed only for the purposes of the loop and is not used elsewhere. When this is the case, it is possible to declare the variable inside the initialization portion of the **for**. For example, here is the preceding program recoded so that the loop control variable **n** is declared as an **int** inside the **for**:

```

// Declare a loop control variable inside the for.
class ForTick {
    public static void main(String args[]) {

        // here, n is declared inside of the for loop
        for(int n=10; n>0; n--)
            System.out.println("tick " + n);
    }
}

```

When you declare a variable inside a **for** loop, there is one important point to remember: the scope of that variable ends when the **for** statement does. (That is, the scope of the variable is limited to the **for** loop.) Outside the **for** loop, the variable will cease to exist. If you need to use the loop control variable elsewhere in your program, you will not be able to declare it inside the **for** loop.

When the loop control variable will not be needed elsewhere, most Java programmers declare it inside the **for**. For example, here is a simple program that tests for prime numbers. Notice that the loop control variable, **i**, is declared inside the **for** since it is not needed elsewhere.

```

// Test for primes.
class FindPrime {
    public static void main(String args[]) {
        int num;
        boolean isPrime;

        num = 14;

        if(num < 2) isPrime = false;
        else isPrime = true;

        for(int i=2; i <= num/i; i++) {
            if((num % i) == 0) {
                isPrime = false;
                break;
            }
        }

        if(isPrime) System.out.println("Prime");
        else System.out.println("Not Prime");
    }
}

```

Using the Comma

There will be times when you will want to include more than one statement in the initialization and iteration portions of the **for** loop. For example, consider the loop in the following program:

```
class Sample {
    public static void main(String args[]) {
        int a, b;

        b = 4;
        for(a=1; a<b; a++) {
            System.out.println("a = " + a);
            System.out.println("b = " + b);
            b--;
        }
    }
}
```

As you can see, the loop is controlled by the interaction of two variables. Since the loop is governed by two variables, it would be useful if both could be included in the **for** statement, itself, instead of **b** being handled manually. Fortunately, Java provides a way to accomplish this. To allow two or more variables to control a **for** loop, Java permits you to include multiple statements in both the initialization and iteration portions of the **for**. Each statement is separated from the next by a comma.

Using the comma, the preceding **for** loop can be more efficiently coded, as shown here:

```
// Using the comma.
class Comma {
    public static void main(String args[]) {
        int a, b;

        for(a=1, b=4; a<b; a++, b--) {
            System.out.println("a = " + a);
            System.out.println("b = " + b);
        }
    }
}
```

In this example, the initialization portion sets the values of both **a** and **b**. The two comma-separated statements in the iteration portion are executed each time the loop repeats. The program generates the following output:

```
a = 1
b = 4
a = 2
b = 3
```

NOTE If you are familiar with C/C++, then you know that in those languages the comma is an operator that can be used in any valid expression. However, this is not the case with Java. In Java, the comma is a separator.

Some for Loop Variations

The **for** loop supports a number of variations that increase its power and applicability. The reason it is so flexible is that its three parts—the initialization, the conditional test, and the iteration—do not need to be used for only those purposes. In fact, the three sections of the **for** can be used for any purpose you desire. Let's look at some examples.

One of the most common variations involves the conditional expression. Specifically, this expression does not need to test the loop control variable against some target value. In fact, the condition controlling the **for** can be any Boolean expression. For example, consider the following fragment:

```
boolean done = false;

for(int i=1; !done; i++) {
    // ...
    if(interrupted()) done = true;
}
```

In this example, the **for** loop continues to run until the **boolean** variable **done** is set to **true**. It does not test the value of **i**.

Here is another interesting **for** loop variation. Either the initialization or the iteration expression or both may be absent, as in this next program:

```
// Parts of the for loop can be empty.
class ForVar {
    public static void main(String args[]) {
        int i;
        boolean done = false;

        i = 0;
        for( ; !done; ) {
            System.out.println("i is " + i);
            if(i == 10) done = true;
            i++;
        }
    }
}
```

Here, the initialization and iteration expressions have been moved out of the **for**. Thus, parts of the **for** are empty. While this is of no value in this simple example—indeed, it would be considered quite poor style—there can be times when this type of approach makes sense. For example, if the initial condition is set through a complex expression elsewhere in the program or if the loop control variable changes in a nonsequential manner determined by actions that occur within the body of the loop, it may be appropriate to leave these parts of the **for** empty.

Here is one more **for** loop variation. You can intentionally create an infinite loop (a loop that never terminates) if you leave all three parts of the **for** empty. For example:

```
for( ; ; ) {
    // ...
}
```

This loop will run forever because there is no condition under which it will terminate. Although there are some programs, such as operating system command processors, that require an infinite loop, most “infinite loops” are really just loops with special termination requirements. As you will soon see, there is a way to terminate a loop—even an infinite loop like the one shown—that does not make use of the normal loop conditional expression.

The For-Each Version of the for Loop

Beginning with JDK 5, a second form of **for** was defined that implements a “for-each” style loop. As you may know, contemporary language theory has embraced the for-each concept, and it has become a standard feature that programmers have come to expect. A for-each style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. Unlike some languages, such as C#, that implement a for-each loop by using the keyword **foreach**, Java adds the for-each capability by enhancing the **for** statement. The advantage of this approach is that no new keyword is required, and no preexisting code is broken. The for-each style of **for** is also referred to as the *enhanced for* loop.

The general form of the for-each version of the **for** is shown here:

for(type itr-var : collection) statement-block

Here, *type* specifies the type and *itr-var* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by *collection*. There are various types of collections that can be used with the **for**, but the only type used in this chapter is the array. (Other types of collections that can be used with the **for**, such as those defined by the Collections Framework, are discussed later in this book.) With each iteration of the loop, the next element in the collection is retrieved and stored in *itr-var*. The loop repeats until all elements in the collection have been obtained.

Because the iteration variable receives values from the collection, *type* must be the same as (or compatible with) the elements stored in the collection. Thus, when iterating over arrays, *type* must be compatible with the element type of the array.

To understand the motivation behind a for-each style loop, consider the type of **for** loop that it is designed to replace. The following fragment uses a traditional **for** loop to compute the sum of the values in an array:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
int sum = 0;  
  
for(int i=0; i < 10; i++) sum += nums[i];
```

To compute the sum, each element in **nums** is read, in order, from start to finish. Thus, the entire array is read in strictly sequential order. This is accomplished by manually indexing the **nums** array by **i**, the loop control variable.

The for-each style **for** automates the preceding loop. Specifically, it eliminates the need to establish a loop counter, specify a starting and ending value, and manually index the array. Instead, it automatically cycles through the entire array, obtaining one element at a time, in

sequence, from beginning to end. For example, here is the preceding fragment rewritten using a for-each version of the **for**:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int x: nums) sum += x;
```

With each pass through the loop, **x** is automatically given a value equal to the next element in **nums**. Thus, on the first iteration, **x** contains 1; on the second iteration, **x** contains 2; and so on. Not only is the syntax streamlined, but it also prevents boundary errors.

Here is an entire program that demonstrates the for-each version of the **for** just described:

```
// Use a for-each style for loop.
class ForEach {
    public static void main(String args[]) {
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int sum = 0;

        // use for-each style for to display and sum the values
        for(int x : nums) {
            System.out.println("Value is: " + x);
            sum += x;
        }

        System.out.println("Summation: " + sum);
    }
}
```

The output from the program is shown here:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55
```

As this output shows, the for-each style **for** automatically cycles through an array in sequence from the lowest index to the highest.

Although the for-each **for** loop iterates until all elements in an array have been examined, it is possible to terminate the loop early by using a **break** statement. For example, this program sums only the first five elements of **nums**:

```
// Use break with a for-each style for.
class ForEach2 {
    public static void main(String args[]) {
        int sum = 0;
```

```

int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// use for to display and sum the values
for(int x : nums) {
    System.out.println("Value is: " + x);
    sum += x;
    if(x == 5) break; // stop the loop when 5 is obtained
}
System.out.println("Summation of first 5 elements: " + sum);
}
}

```

This is the output produced:

```

Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Summation of first 5 elements: 15

```

As is evident, the **for** loop stops after the fifth element has been obtained. The **break** statement can also be used with Java's other loops, and it is discussed in detail later in this chapter.

There is one important point to understand about the for-each style loop. Its iteration variable is “read-only” as it relates to the underlying array. An assignment to the iteration variable has no effect on the underlying array. In other words, you can't change the contents of the array by assigning the iteration variable a new value. For example, consider this program:

```

// The for-each loop is essentially read-only.
class NoChange {
    public static void main(String args[]) {
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        for(int x: nums) {
            System.out.print(x + " ");
            x = x * 10; // no effect on nums
        }

        System.out.println();

        for(int x : nums)
            System.out.print(x + " ");

        System.out.println();
    }
}

```

The first **for** loop increases the value of the iteration variable by a factor of 10. However, this assignment has no effect on the underlying array **nums**, as the second **for** loop illustrates. The output, shown here, proves this point:

```

1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10

```

Iterating Over Multidimensional Arrays

The enhanced version of the **for** also works on multidimensional arrays. Remember, however, that in Java, multidimensional arrays consist of *arrays of arrays*. (For example, a two-dimensional array is an array of one-dimensional arrays.) This is important when iterating over a multidimensional array, because each iteration obtains the *next array*, not an individual element. Furthermore, the iteration variable in the **for** loop must be compatible with the type of array being obtained. For example, in the case of a two-dimensional array, the iteration variable must be a reference to a one-dimensional array. In general, when using the for-each **for** to iterate over an array of N dimensions, the objects obtained will be arrays of $N-1$ dimensions. To understand the implications of this, consider the following program. It uses nested **for** loops to obtain the elements of a two-dimensional array in row-order, from first to last.

```
// Use for-each style for on a two-dimensional array.
class ForEach3 {
    public static void main(String args[]) {
        int sum = 0;
        int nums[][] = new int[3][5];

        // give nums some values
        for(int i = 0; i < 3; i++)
            for(int j = 0; j < 5; j++)
                nums[i][j] = (i+1)*(j+1);

        // use for-each for to display and sum the values
        for(int x[] : nums) {
            for(int y : x) {
                System.out.println("Value is: " + y);
                sum += y;
            }
        }
        System.out.println("Summation: " + sum);
    }
}
```

The output from this program is shown here:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
```

```
Value is: 12
Value is: 15
Summation: 90
```

In the program, pay special attention to this line:

```
for(int x[] : nums) {
```

Notice how **x** is declared. It is a reference to a one-dimensional array of integers. This is necessary because each iteration of the **for** obtains the next *array* in **nums**, beginning with the array specified by **nums[0]**. The inner **for** loop then cycles through each of these arrays, displaying the values of each element.

Applying the Enhanced for

Since the for-each style **for** can only cycle through an array sequentially, from start to finish, you might think that its use is limited, but this is not true. A large number of algorithms require exactly this mechanism. One of the most common is searching. For example, the following program uses a **for** loop to search an unsorted array for a value. It stops if the value is found.

```
// Search an array using for-each style for.
class Search {
    public static void main(String args[]) {
        int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
        int val = 5;
        boolean found = false;

        // use for-each style for to search nums for val
        for(int x : nums) {
            if(x == val) {
                found = true;
                break;
            }
        }

        if(found)
            System.out.println("Value found!");
    }
}
```

The for-each style **for** is an excellent choice in this application because searching an unsorted array involves examining each element in sequence. (Of course, if the array were sorted, a binary search could be used, which would require a different style loop.) Other types of applications that benefit from for-each style loops include computing an average, finding the minimum or maximum of a set, looking for duplicates, and so on.

Although we have been using arrays in the examples in this chapter, the for-each style **for** is especially useful when operating on collections defined by the Collections Framework, which is described in Part II. More generally, the **for** can cycle through the elements of any collection of objects, as long as that collection satisfies a certain set of constraints, which are described in Chapter 18.

Nested Loops

Like all other programming languages, Java allows loops to be nested. That is, one loop may be inside another. For example, here is a program that nests **for** loops:

```
// Loops may be nested.
class Nested {
    public static void main(String args[]) {
        int i, j;
        for(i=0; i<10; i++) {
            for(j=i; j<10; j++)
                System.out.print(".");
            System.out.println();
        }
    }
}
```

The output produced by this program is shown here:

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

Jump Statements

Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control to another part of your program. Each is examined here.

NOTE In addition to the jump statements discussed here, Java supports one other way that you can change your program's flow of execution: through exception handling. Exception handling provides a structured method by which run-time errors can be trapped and handled by your program. It is supported by the keywords **try**, **catch**, **throw**, **throws**, and **finally**. In essence, the exception handling mechanism allows your program to perform a nonlocal branch. Since exception handling is a large topic, it is discussed in its own chapter, Chapter 10.

Using break

In Java, the **break** statement has three uses. First, as you have seen, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop. Third, it can be used as a “civilized” form of `goto`. The last two uses are explained here.

Using break to Exit a Loop

By using **break**, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a **break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. Here is a simple example:

```
// Using break to exit a loop.
class BreakLoop {
    public static void main(String args[]) {
        for(int i=0; i<100; i++) {
            if(i == 10) break; // terminate loop if i is 10
            System.out.println("i: " + i);
        }
        System.out.println("Loop complete.");
    }
}
```

This program generates the following output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

As you can see, although the **for** loop is designed to run from 0 to 99, the **break** statement causes it to terminate early, when **i** equals 10.

The **break** statement can be used with any of Java's loops, including intentionally infinite loops. For example, here is the preceding program coded by use of a **while** loop. The output from this program is the same as just shown.

```
// Using break to exit a while loop.
class BreakLoop2 {
    public static void main(String args[]) {
        int i = 0;

        while(i < 100) {
            if(i == 10) break; // terminate loop if i is 10
            System.out.println("i: " + i);
            i++;
        }
        System.out.println("Loop complete.");
    }
}
```

When used inside a set of nested loops, the **break** statement will only break out of the innermost loop. For example:

```
// Using break with nested loops.
class BreakLoop3 {
    public static void main(String args[]) {
        for(int i=0; i<3; i++) {
            System.out.print("Pass " + i + ": ");
            for(int j=0; j<100; j++) {
                if(j == 10) break; // terminate loop if j is 10
                System.out.print(j + " ");
            }
            System.out.println();
        }
        System.out.println("Loops complete.");
    }
}
```

This program generates the following output:

```
Pass 0: 0 1 2 3 4 5 6 7 8 9
Pass 1: 0 1 2 3 4 5 6 7 8 9
Pass 2: 0 1 2 3 4 5 6 7 8 9
Loops complete.
```

As you can see, the **break** statement in the inner loop only causes termination of that loop. The outer loop is unaffected.

Here are two other points to remember about **break**. First, more than one **break** statement may appear in a loop. However, be careful. Too many **break** statements have the tendency to destructure your code. Second, the **break** that terminates a **switch** statement affects only that **switch** statement and not any enclosing loops.

REMEMBER **break** was not designed to provide the normal means by which a loop is terminated. The loop's conditional expression serves this purpose. The **break** statement should be used to cancel a loop only when some sort of special situation occurs.

Using **break** as a Form of Goto

In addition to its uses with the **switch** statement and loops, the **break** statement can also be employed by itself to provide a “civilized” form of the goto statement. Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner. This usually makes goto-ridden code hard to understand and hard to maintain. It also prohibits certain compiler optimizations. There are, however, a few places where the goto is a valuable and legitimate construct for flow control. For example, the goto can be useful when you are exiting from a deeply nested set of loops. To handle such situations, Java defines an expanded form of the **break** statement. By using this form of **break**, you can, for example, break out of one or more blocks of code. These blocks need not be part of a loop or a **switch**. They can be any block. Further, you can specify precisely where execution will resume, because this form of **break** works with a label. As you will see, **break** gives you the benefits of a goto without its problems.

The general form of the labeled **break** statement is shown here:

```
break label;
```

Most often, *label* is the name of a label that identifies a block of code. This can be a stand-alone block of code but it can also be a block that is the target of another statement. When this form of **break** executes, control is transferred out of the named block. The labeled block must enclose the **break** statement, but it does not need to be the immediately enclosing block. This means, for example, that you can use a labeled **break** statement to exit from a set of nested blocks. But you cannot use **break** to transfer control out of a block that does not enclose the **break** statement.

To name a block, put a label at the start of it. A *label* is any valid Java identifier followed by a colon. Once you have labeled a block, you can then use this label as the target of a **break** statement. Doing so causes execution to resume at the *end* of the labeled block. For example, the following program shows three nested blocks, each with its own label. The **break** statement causes execution to jump forward, past the end of the block labeled **second**, skipping the two **println()** statements.

```
// Using break as a civilized form of goto.
class Break {
    public static void main(String args[]) {
        boolean t = true;

        first: {
            second: {
                third: {
                    System.out.println("Before the break.");
                    if(t) break second; // break out of second block
                    System.out.println("This won't execute");
                }
                System.out.println("This won't execute");
            }
            System.out.println("This is after second block.");
        }
    }
}
```

Running this program generates the following output:

```
Before the break.
This is after second block.
```

One of the most common uses for a labeled **break** statement is to exit from nested loops. For example, in the following program, the outer loop executes only once:

```
// Using break to exit from nested loops
class BreakLoop4 {
    public static void main(String args[]) {
        outer: for(int i=0; i<3; i++) {
            System.out.print("Pass " + i + ": ");
            for(int j=0; j<100; j++) {
                if(j == 10) break outer; // exit both loops
            }
        }
    }
}
```

```
        System.out.print(j + " ");
    }
    System.out.println("This will not print");
}
System.out.println("Loops complete.");
}
```

This program generates the following output:

Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete.

As you can see, when the inner loop breaks to the outer loop, both loops have been terminated. Notice that this example labels the **for** statement, which has a block of code as its target.

Keep in mind that you cannot break to any label which is not defined for an enclosing block. For example, the following program is invalid and will not compile:

```
// This program contains an error.
class BreakErr {
    public static void main(String args[]) {
        one: for(int i=0; i<3; i++) {
            System.out.print("Pass " + i + ": ");
        }
        for(int j=0; j<100; j++) {
            if(j == 10) break one; // WRONG
            System.out.print(j + " ");
        }
    }
}
```

Since the loop labeled **one** does not enclose the **break** statement, it is not possible to transfer control out of that block.

Using continue

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The **continue** statement performs such an action. In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop. In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

Here is an example program that uses **continue** to cause two numbers to be printed on each line:

```
// Demonstrate continue.  
class Continue {  
    public static void main(String args[]) {
```

```

        for(int i=0; i<10; i++) {
            System.out.print(i + " ");
            if (i%2 == 0) continue;
            System.out.println("");
        }
    }
}

```

This code uses the `%` operator to check if `i` is even. If it is, the loop continues without printing a newline. Here is the output from this program:

```

0 1
2 3
4 5
6 7
8 9

```

As with the `break` statement, `continue` may specify a label to describe which enclosing loop to continue. Here is an example program that uses `continue` to print a triangular multiplication table for 0 through 9:

```

// Using continue with a label.
class ContinueLabel {
    public static void main(String args[]) {
outer: for (int i=0; i<10; i++) {
    for(int j=0; j<10; j++) {
        if(j > i) {
            System.out.println();
            continue outer;
        }
        System.out.print(" " + (i * j));
    }
    System.out.println();
}
}

```

The `continue` statement in this example terminates the loop counting `j` and continues with the next iteration of the loop counting `i`. Here is the output of this program:

```

0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81

```

Good uses of `continue` are rare. One reason is that Java provides a rich set of loop statements which fit most applications. However, for those special circumstances in which early iteration is needed, the `continue` statement provides a structured way to accomplish it.

return

The last control statement is **return**. The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement. Although a full discussion of **return** must wait until methods are discussed in Chapter 6, a brief look at **return** is presented here.

At any time in a method, the **return** statement can be used to cause execution to branch back to the caller of the method. Thus, the **return** statement immediately terminates the method in which it is executed. The following example illustrates this point. Here, **return** causes execution to return to the Java run-time system, since it is the run-time system that calls **main()**:

```
// Demonstrate return.
class Return {
    public static void main(String args[]) {
        boolean t = true;

        System.out.println("Before the return.");

        if(t) return; // return to caller

        System.out.println("This won't execute.");
    }
}
```

The output from this program is shown here:

```
Before the return.
```

As you can see, the final **println()** statement is not executed. As soon as **return** is executed, control passes back to the caller.

One last point: In the preceding program, the **if(t)** statement is necessary. Without it, the Java compiler would flag an “unreachable code” error because the compiler would know that the last **println()** statement would never be executed. To prevent this error, the **if** statement is used here to trick the compiler for the sake of this demonstration.

CHAPTER

6

Introducing Classes

The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such, the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class.

Because the class is so fundamental to Java, this and the next few chapters will be devoted to it. Here, you will be introduced to the basic elements of a class and learn how a class can be used to create objects. You will also learn about methods, constructors, and the **this** keyword.

Class Fundamentals

Classes have been used since the beginning of this book. However, until now, only the most rudimentary form of a class has been shown. The classes created in the preceding chapters primarily exist simply to encapsulate the **main()** method, which has been used to demonstrate the basics of the Java syntax. As you will see, classes are substantially more powerful than the limited ones presented so far.

Perhaps the most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a *template* for an object, and an object is an *instance* of a class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.

The General Form of a Class

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both. As you will see, a class' code defines the interface to its data.

A class is declared by use of the **class** keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a **class** definition is shown here:

```
class classname {  
    type instance-variable1;
```

```

type instance-variable2;
// ...
type instance-variableN;

type methodname1(parameter-list) {
    // body of method
}
type methodname2(parameter-list) {
    // body of method
}
// ...
type methodnameN(parameter-list) {
    // body of method
}
}

```

The data, or variables, defined within a **class** are called *instance variables*. The code is contained within *methods*. Collectively, the methods and variables defined within a class are called *members* of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, as a general rule, it is the methods that determine how a class' data can be used.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. We will come back to this point shortly, but it is an important concept to learn early.

All methods have the same general form as **main()**, which we have been using thus far. However, most methods will not be specified as **static** or **public**. Notice that the general form of a class does not specify a **main()** method. Java classes do not need to have a **main()** method. You only specify one if that class is the starting point for your program. Further, some kinds of Java applications, such as applets, don't require a **main()** method at all.

A Simple Class

Let's begin our study of the class with a simple example. Here is a class called **Box** that defines three instance variables: **width**, **height**, and **depth**. Currently, **Box** does not contain any methods (but some will be added soon).

```

class Box {
    double width;
    double height;
    double depth;
}

```

As stated, a class defines a new type of data. In this case, the new data type is called **Box**. You will use this name to declare objects of type **Box**. It is important to remember that a class declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any objects of type **Box** to come into existence.

To actually create a **Box** object, you will use a statement like the following:

```
Box mybox = new Box(); // create a Box object called mybox
```

After this statement executes, **mybox** will be an instance of **Box**. Thus, it will have “physical” reality. For the moment, don’t worry about the details of this statement.

As mentioned earlier, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every **Box** object will contain its own copies of the instance variables **width**, **height**, and **depth**. To access these variables, you will use the *dot* (.) operator. The dot operator links the name of the object with the name of an instance variable. For example, to assign the **width** variable of **mybox** the value 100, you would use the following statement:

```
mybox.width = 100;
```

This statement tells the compiler to assign the copy of **width** that is contained within the **mybox** object the value of 100. In general, you use the dot operator to access both the instance variables and the methods within an object. One other point: Although commonly referred to as the dot *operator*, the formal specification for Java categorizes the . as a separator. However, since the use of the term “dot operator” is widespread, it is used in this book.

Here is a complete program that uses the **Box** class:

```
/* A program that uses the Box class.

   Call this file BoxDemo.java
*/
class Box {
    double width;
    double height;
    double depth;
}

// This class declares an object of type Box.
class BoxDemo {
    public static void main(String args[]) {
        Box mybox = new Box();
        double vol;

        // assign values to mybox's instance variables
        mybox.width = 10;
        mybox.height = 20;
        mybox.depth = 15;

        // compute volume of box
        vol = mybox.width * mybox.height * mybox.depth;

        System.out.println("Volume is " + vol);
    }
}
```

You should call the file that contains this program **BoxDemo.java**, because the **main()** method is in the class called **BoxDemo**, not the class called **Box**. When you compile this

program, you will find that two **.class** files have been created, one for **Box** and one for **BoxDemo**. The Java compiler automatically puts each class into its own **.class** file. It is not necessary for both the **Box** and the **BoxDemo** class to actually be in the same source file. You could put each class in its own file, called **Box.java** and **BoxDemo.java**, respectively.

To run this program, you must execute **BoxDemo.class**. When you do, you will see the following output:

```
Volume is 3000.0
```

As stated earlier, each object has its own copies of the instance variables. This means that if you have two **Box** objects, each has its own copy of **depth**, **width**, and **height**. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another. For example, the following program declares two **Box** objects:

```
// This program declares two Box objects.

class Box {
    double width;
    double height;
    double depth;
}

class BoxDemo2 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* assign different values to mybox2's
           instance variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // compute volume of first box
        vol = mybox1.width * mybox1.height * mybox1.depth;
        System.out.println("Volume is " + vol);

        // compute volume of second box
        vol = mybox2.width * mybox2.height * mybox2.depth;
        System.out.println("Volume is " + vol);
    }
}
```

The output produced by this program is shown here:

```
Volume is 3000.0  
Volume is 162.0
```

As you can see, **mybox1**'s data is completely separate from the data contained in **mybox2**.

Declaring Objects

As just explained, when you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator. The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let's look at the details of this procedure.

In the preceding sample programs, a line similar to the following is used to declare an object of type **Box**:

```
Box mybox = new Box();
```

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object  
mybox = new Box(); // allocate a Box object
```

The first line declares **mybox** as a reference to an object of type **Box**. At this point, **mybox** does not yet refer to an actual object. The next line allocates an object and assigns a reference to it to **mybox**. After the second line executes, you can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds, in essence, the memory address of the actual **Box** object. The effect of these two lines of code is depicted in Figure 6-1.

NOTE Those readers familiar with C/C++ have probably noticed that object references appear to be similar to pointers. This suspicion is, essentially, correct. An object reference is similar to a memory pointer. The main difference—and the key to Java's safety—is that you cannot manipulate references as you can actual pointers. Thus, you cannot cause an object reference to point to an arbitrary memory location or manipulate it like an integer.

A Closer Look at **new**

As just explained, the **new** operator dynamically allocates memory for an object. It has this general form:

```
class-var = new classname( );
```

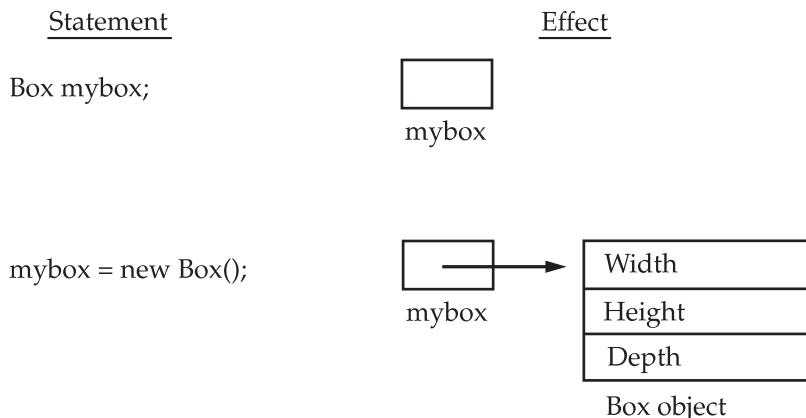


Figure 6-1 Declaring an object of type **Box**

Here, *class-var* is a variable of the class type being created. The *classname* is the name of the class that is being instantiated. The class name followed by parentheses specifies the *constructor* for the class. A constructor defines what occurs when an object of a class is created. Constructors are an important part of all classes and have many significant attributes. Most real-world classes explicitly define their own constructors within their class definition. However, if no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with **Box**. For now, we will use the default constructor. Soon, you will see how to define your own constructors.

At this point, you might be wondering why you do not need to use **new** for such things as integers or characters. The answer is that Java's primitive types are not implemented as objects. Rather, they are implemented as "normal" variables. This is done in the interest of efficiency. As you will see, objects have many features and attributes that require Java to treat them differently than it treats the primitive types. By not applying the same overhead to the primitive types that applies to objects, Java can implement the primitive types more efficiently. Later, you will see object versions of the primitive types that are available for your use in those situations in which complete objects of these types are needed.

It is important to understand that **new** allocates memory for an object during run time. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program. However, since memory is finite, it is possible that **new** will not be able to allocate memory for an object because insufficient memory exists. If this happens, a run-time exception will occur. (You will learn how to handle exceptions in Chapter 10.) For the sample programs in this book, you won't need to worry about running out of memory, but you will need to consider this possibility in real-world programs that you write.

Let's once again review the distinction between a class and an object. A class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class. Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.) It is important to keep this distinction clearly in mind.

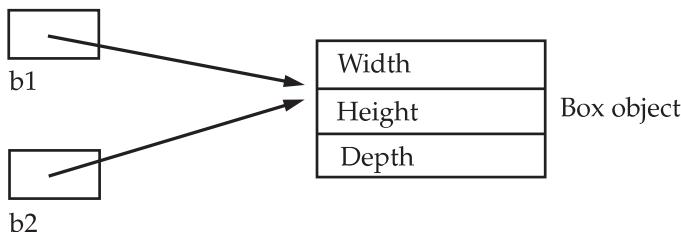
Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

```
Box b1 = new Box();
Box b2 = b1;
```

You might think that **b2** is being assigned a reference to a copy of the object referred to by **b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the *same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.

This situation is depicted here:



Although **b1** and **b2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**. For example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, **b1** has been set to **null**, but **b2** still points to the original object.

REMEMBER When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

Introducing Methods

As mentioned at the beginning of this chapter, classes usually consist of two things: instance variables and methods. The topic of methods is a large one because Java gives them so much power and flexibility. In fact, much of the next chapter is devoted to methods. However, there are some fundamentals that you need to learn now so that you can begin to add methods to your classes.

This is the general form of a method:

```
type name(parameter-list) {
    // body of method
}
```

Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**. The name of the method is specified by *name*. This can be any legal identifier other than those already used by other items within the current scope. The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.

Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

```
return value;
```

Here, *value* is the value returned.

In the next few sections, you will see how to create various types of methods, including those that take parameters and those that return values.

Adding a Method to the Box Class

Although it is perfectly fine to create a class that contains only data, it rarely happens. Most of the time, you will use methods to access the instance variables defined by the class. In fact, methods define the interface to most classes. This allows the class implementor to hide the specific layout of internal data structures behind cleaner method abstractions. In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself.

Let's begin by adding a method to the **Box** class. It may have occurred to you while looking at the preceding programs that the computation of a box's volume was something that was best handled by the **Box** class rather than the **BoxDemo** class. After all, since the volume of a box is dependent upon the size of the box, it makes sense to have the **Box** class compute it. To do this, you must add a method to **Box**, as shown here:

```
// This program includes a method inside the box class.

class Box {
    double width;
    double height;
    double depth;

    // display volume of a box
    void volume() {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);
    }
}

class BoxDemo3 {
    public static void main(String args[]) {
```

```
Box mybox1 = new Box();
Box mybox2 = new Box();

// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;

/* assign different values to mybox2's
   instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;

// display volume of first box
mybox1.volume();

// display volume of second box
mybox2.volume();
}
```

This program generates the following output, which is the same as the previous version.

```
Volume is 3000.0
Volume is 162.0
```

Look closely at the following two lines of code:

```
mybox1.volume();
mybox2.volume();
```

The first line here invokes the **volume()** method on **mybox1**. That is, it calls **volume()** relative to the **mybox1** object, using the object's name followed by the dot operator. Thus, the call to **mybox1.volume()** displays the volume of the box defined by **mybox1**, and the call to **mybox2.volume()** displays the volume of the box defined by **mybox2**. Each time **volume()** is invoked, it displays the volume for the specified box.

If you are unfamiliar with the concept of calling a method, the following discussion will help clear things up. When **mybox1.volume()** is executed, the Java run-time system transfers control to the code defined inside **volume()**. After the statements inside **volume()** have executed, control is returned to the calling routine, and execution resumes with the line of code following the call. In the most general sense, a method is Java's way of implementing subroutines.

There is something very important to notice inside the **volume()** method: the instance variables **width**, **height**, and **depth** are referred to directly, without preceding them with an object name or the dot operator. When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator. This is easy to understand if you think about it. A method is always invoked relative to some object of its class. Once this invocation has occurred, the object is known. Thus, within a method, there is no need to specify the object a second time. This means that **width**, **height**, and **depth** inside **volume()** implicitly refer to the copies of those variables found in the object that invokes **volume()**.

Let's review: When an instance variable is accessed by code that is not part of the class in which that instance variable is defined, it must be done through an object, by use of the dot operator. However, when an instance variable is accessed by code that is part of the same class as the instance variable, that variable can be referred to directly. The same thing applies to methods.

Returning a Value

While the implementation of `volume()` does move the computation of a box's volume inside the `Box` class where it belongs, it is not the best way to do it. For example, what if another part of your program wanted to know the volume of a box, but not display its value? A better way to implement `volume()` is to have it compute the volume of the box and return the result to the caller. The following example, an improved version of the preceding program, does just that:

```
// Now, volume() returns the volume of a box.

class Box {
    double width;
    double height;
    double depth;

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo4 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* assign different values to mybox2's
           instance variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

As you can see, when **volume()** is called, it is put on the right side of an assignment statement. On the left is a variable, in this case **vol**, that will receive the value returned by **volume()**. Thus, after

```
vol = mybox1.volume();
```

executes, the value of **mybox1.volume()** is 3,000 and this value then is stored in **vol**.

There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean**, you could not return an integer.
- The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method.

One more point: The preceding program can be written a bit more efficiently because there is actually no need for the **vol** variable. The call to **volume()** could have been used in the **println()** statement directly, as shown here:

```
System.out.println("Volume is " + mybox1.volume());
```

In this case, when **println()** is executed, **mybox1.volume()** will be called automatically and its value will be passed to **println()**.

Adding a Method That Takes Parameters

While some methods don't need parameters, most do. Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations. To illustrate this point, let's use a very simple example. Here is a method that returns the square of the number 10:

```
int square()
{
    return 10 * 10;
}
```

While this method does, indeed, return the value of 10 squared, its use is very limited. However, if you modify the method so that it takes a parameter, as shown next, then you can make **square()** much more useful.

```
int square(int i)
{
    return i * i;
}
```

Now, **square()** will return the square of whatever value it is called with. That is, **square()** is now a general-purpose method that can compute the square of any integer value, rather than just 10.

Here is an example:

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
```

```
y = 2;
x = square(y); // x equals 4
```

In the first call to **square()**, the value 5 will be passed into parameter **i**. In the second call, **i** will receive the value 9. The third invocation passes the value of **y**, which is 2 in this example. As these examples show, **square()** is able to return the square of whatever data it is passed.

It is important to keep the two terms *parameter* and *argument* straight. A *parameter* is a variable defined by a method that receives a value when the method is called. For example, in **square()**, **i** is a parameter. An *argument* is a value that is passed to a method when it is invoked. For example, **square(100)** passes 100 as an argument. Inside **square()**, the parameter **i** receives that value.

You can use a parameterized method to improve the **Box** class. In the preceding examples, the dimensions of each box had to be set separately by use of a sequence of statements, such as:

```
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
```

While this code works, it is troubling for two reasons. First, it is clumsy and error prone. For example, it would be easy to forget to set a dimension. Second, in well-designed Java programs, instance variables should be accessed only through methods defined by their class. In the future, you can change the behavior of a method, but you can't change the behavior of an exposed instance variable.

Thus, a better approach to setting the dimensions of a box is to create a method that takes the dimensions of a box in its parameters and sets each instance variable appropriately. This concept is implemented by the following program:

```
// This program uses a parameterized method.

class Box {
    double width;
    double height;
    double depth;

    // compute and return volume
    double volume() {
        return width * height * depth;
    }

    // sets dimensions of box
    void setDim(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
}

class BoxDemo5 {
```

```
public static void main(String args[]) {  
    Box mybox1 = new Box();  
    Box mybox2 = new Box();  
    double vol;  
  
    // initialize each box  
    mybox1.setDim(10, 20, 15);  
    mybox2.setDim(3, 6, 9);  
  
    // get volume of first box  
    vol = mybox1.volume();  
    System.out.println("Volume is " + vol);  
  
    // get volume of second box  
    vol = mybox2.volume();  
    System.out.println("Volume is " + vol);  
}
```

As you can see, the **setDim()** method is used to set the dimensions of each box. For example, when

```
mybox1.setDim(10, 20, 15);
```

is executed, 10 is copied into parameter **w**, 20 is copied into **h**, and 15 is copied into **d**. Inside **setDim()** the values of **w**, **h**, and **d** are then assigned to **width**, **height**, and **depth**, respectively.

For many readers, the concepts presented in the preceding sections will be familiar. However, if such things as method calls, arguments, and parameters are new to you, then you might want to take some time to experiment before moving on. The concepts of the method invocation, parameters, and return values are fundamental to Java programming.

Constructors

It can be tedious to initialize all of the variables in a class each time an instance is created. Even when you add convenience functions like **setDim()**, it would be simpler and more concise to have all of the setup done at the time the object is first created. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.

A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called when the object is created, before the **new** operator completes. Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

You can rework the **Box** example so that the dimensions of a box are automatically initialized when an object is constructed. To do so, replace **setDim()** with a constructor.

Let's begin by defining a simple constructor that simply sets the dimensions of each box to the same values. This version is shown here:

```
/* Here, Box uses a constructor to initialize the
   dimensions of a box.
*/
class Box {
    double width;
    double height;
    double depth;

    // This is the constructor for Box.
    Box() {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo6 {
    public static void main(String args[]) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

When this program is run, it generates the following results:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

As you can see, both **mybox1** and **mybox2** were initialized by the **Box()** constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both **mybox1** and **mybox2** will have the same volume. The **println()** statement

inside **Box()** is for the sake of illustration only. Most constructors will not display anything. They will simply initialize an object.

Before moving on, let's reexamine the **new** operator. As you know, when you allocate an object, you use the following general form:

```
class-var = new classname( );
```

Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line

```
Box mybox1 = new Box();
```

new Box() is calling the **Box()** constructor. When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. This is why the preceding line of code worked in earlier versions of **Box** that did not define a constructor. The default constructor automatically initializes all instance variables to their default values, which are zero, **null**, and **false**, for numeric types, reference types, and **boolean**, respectively. The default constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones. Once you define your own constructor, the default constructor is no longer used.

Parameterized Constructors

While the **Box()** constructor in the preceding example does initialize a **Box** object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct **Box** objects of various dimensions. The easy solution is to add parameters to the constructor. As you can probably guess, this makes it much more useful. For example, the following version of **Box** defines a parameterized constructor that sets the dimensions of a box as specified by those parameters. Pay special attention to how **Box** objects are created.

```
/* Here, Box uses a parameterized constructor to
   initialize the dimensions of a box.
*/
class Box {
    double width;
    double height;
    double depth;

    // This is the constructor for Box.
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}
```

```

class BoxDemo7 {
    public static void main(String args[]) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box(3, 6, 9);

        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}

```

The output from this program is shown here:

```

Volume is 3000.0
Volume is 162.0

```

As you can see, each object is initialized as specified in the parameters to its constructor. For example, in the following line,

```
Box mybox1 = new Box(10, 20, 15);
```

the values 10, 20, and 15 are passed to the **Box()** constructor when **new** creates the object. Thus, **mybox1**'s copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15, respectively.

The **this** Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class' type is permitted.

To better understand what **this** refers to, consider the following version of **Box()**:

```

// A redundant use of this.
Box(double w, double h, double d) {
    this.width = w;
    this.height = h;
    this.depth = d;
}

```

This version of **Box()** operates exactly like the earlier version. The use of **this** is redundant, but perfectly correct. Inside **Box()**, **this** will always refer to the invoking object. While it is

redundant in this case, **this** is useful in other contexts, one of which is explained in the next section.

Instance Variable Hiding

As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes. Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable. This is why **width**, **height**, and **depth** were not used as the names of the parameters to the **Box()** constructor inside the **Box** class. If they had been, then **width**, for example, would have referred to the formal parameter, hiding the instance variable **width**. While it is usually easier to simply use different names, there is another way around this situation. Because **this** lets you refer directly to the object, you can use it to resolve any namespace collisions that might occur between instance variables and local variables. For example, here is another version of **Box()**, which uses **width**, **height**, and **depth** for parameter names and then uses **this** to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.  
Box(double width, double height, double depth) {  
    this.width = width;  
    this.height = height;  
    this.depth = depth;  
}
```

A word of caution: The use of **this** in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables. Of course, other programmers believe the contrary—that it is a good convention to use the same names for clarity, and use **this** to overcome the instance variable hiding. It is a matter of taste which approach you adopt.

Garbage Collection

Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection*. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

The finalize() Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization*. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the **finalize()** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize()** method, you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize()** method on the object.

The **finalize()** method has this general form:

```
protected void finalize( )
{
    // finalization code here
}
```

Here, the keyword **protected** is a specifier that limits access to **finalize()**. This and the other access modifiers are explained in Chapter 7.

It is important to understand that **finalize()** is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—**finalize()** will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize()** for normal program operation.

NOTE If you are familiar with C++, then you know that C++ allows you to define a destructor for a class, which is called when an object goes out-of-scope. Java does not support this idea or provide for destructors. The **finalize()** method only approximates the function of a destructor. As you get more experienced with Java, you will see that the need for destructor functions is minimal because of Java's garbage collection subsystem.

A Stack Class

While the **Box** class is useful to illustrate the essential elements of a class, it is of little practical value. To show the real power of classes, this chapter will conclude with a more sophisticated example. As you recall from the discussion of object-oriented programming (OOP) presented in Chapter 2, one of OOP's most important benefits is the encapsulation of data and the code that manipulates that data. As you have seen, the class is the mechanism by which encapsulation is achieved in Java. By creating a class, you are creating a new data type that defines both the nature of the data being manipulated and the routines used to manipulate it. Further, the methods define a consistent and controlled interface to the class' data. Thus, you can use the class through its methods without having to worry about the details of its implementation or how the data is actually managed within the class. In a sense, a class is like a “data engine.” No knowledge of what goes on inside the engine is required to use the engine through its controls. In fact, since the details are hidden, its

inner workings can be changed as needed. As long as your code uses the class through its methods, internal details can change without causing side effects outside the class.

To see a practical application of the preceding discussion, let's develop one of the archetypal examples of encapsulation: the stack. A *stack* stores data using first-in, last-out ordering. That is, a stack is like a stack of plates on a table—the first plate put down on the table is the last plate to be used. Stacks are controlled through two operations traditionally called *push* and *pop*. To put an item on top of the stack, you will use *push*. To take an item off the stack, you will use *pop*. As you will see, it is easy to encapsulate the entire stack mechanism.

Here is a class called **Stack** that implements a stack for up to ten integers:

```
// This class defines an integer stack that can hold 10 values
class Stack {
    int stck[] = new int[10];
    int tos;

    // Initialize top-of-stack
    Stack() {
        tos = -1;
    }

    // Push an item onto the stack
    void push(int item) {
        if(tos==9)
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }

    // Pop an item from the stack
    int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}
```

As you can see, the **Stack** class defines two data items and three methods. The stack of integers is held by the array **stck**. This array is indexed by the variable **tos**, which always contains the index of the top of the stack. The **Stack()** constructor initializes **tos** to **-1**, which indicates an empty stack. The method **push()** puts an item on the stack. To retrieve an item, call **pop()**. Since access to the stack is through **push()** and **pop()**, the fact that the stack is held in an array is actually not relevant to using the stack. For example, the stack could be held in a more complicated data structure, such as a linked list, yet the interface defined by **push()** and **pop()** would remain the same.

The class **TestStack**, shown here, demonstrates the **Stack** class. It creates two integer stacks, pushes some values onto each, and then pops them off.

```
class TestStack {
    public static void main(String args[]) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();

        // push some numbers onto the stack
        for(int i=0; i<10; i++) mystack1.push(i);
        for(int i=10; i<20; i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<10; i++)
            System.out.println(mystack1.pop());

        System.out.println("Stack in mystack2:");
        for(int i=0; i<10; i++)
            System.out.println(mystack2.pop());
    }
}
```

This program generates the following output:

```
Stack in mystack1:
9
8
7
6
5
4
3
2
1
0
Stack in mystack2:
19
18
17
16
15
14
13
12
11
10
```

As you can see, the contents of each stack are separate.

One last point about the **Stack** class. As it is currently implemented, it is possible for the array that holds the stack, **stck**, to be altered by code outside of the **Stack** class. This leaves **Stack** open to misuse or mischief. In the next chapter, you will see how to remedy this situation.

CHAPTER

7

A Closer Look at Methods and Classes

This chapter continues the discussion of methods and classes begun in the preceding chapter. It examines several topics relating to methods, including overloading, parameter passing, and recursion. The chapter then returns to the class, discussing access control, the use of the keyword **static**, and one of Java's most important built-in classes: **String**.

Overloading Methods

In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as *method overloading*. Method overloading is one of the ways that Java supports polymorphism. If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. But as you will see, method overloading is one of Java's most exciting and useful features.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }

    // Overload test for one integer parameter.
    void test(int a) {
        System.out.println("a: " + a);
    }
}
```

```

// Overload test for two integer parameters.
void test(int a, int b) {
    System.out.println("a and b: " + a + " " + b);
}

// Overload test for a double parameter
double test(double a) {
    System.out.println("double a: " + a);
    return a*a;
}
}

class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        double result;

        // call all versions of test()
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25): " + result);
    }
}

```

This program generates the following output:

```

No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625

```

As you can see, `test()` is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one **double** parameter. The fact that the fourth version of `test()` also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters. However, this match need not always be exact. In some cases, Java's automatic type conversions can play a role in overload resolution. For example, consider the following program:

```

// Automatic type conversions apply to overloading.
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }

    // Overload test for two integer parameters.
    void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
    }
}

```

```
}

// Overload test for a double parameter
void test(double a) {
    System.out.println("Inside test(double) a: " + a);
}
}

class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        int i = 88;

        ob.test();
        ob.test(10, 20);

        ob.test(i); // this will invoke test(double)
        ob.test(123.2); // this will invoke test(double)
    }
}
```

This program generates the following output:

```
No parameters
a and b: 10 20
Inside test(double) a: 88
Inside test(double) a: 123.2
```

As you can see, this version of **OverloadDemo** does not define **test(int)**. Therefore, when **test()** is called with an integer argument inside **Overload**, no matching method is found. However, Java can automatically convert an integer into a **double**, and this conversion can be used to resolve the call. Therefore, after **test(int)** is not found, Java elevates **i** to **double** and then calls **test(double)**. Of course, if **test(int)** had been defined, it would have been called instead. Java will employ its automatic type conversions only if no exact match is found.

Method overloading supports polymorphism because it is one way that Java implements the “one interface, multiple methods” paradigm. To understand how, consider the following. In languages that do not support method overloading, each method must be given a unique name. However, frequently you will want to implement essentially the same method for different types of data. Consider the absolute value function. In languages that do not support overloading, there are usually three or more versions of this function, each with a slightly different name. For instance, in C, the function **abs()** returns the absolute value of an integer, **labs()** returns the absolute value of a long integer, and **fabs()** returns the absolute value of a floating-point value. Since C does not support overloading, each function has its own name, even though all three functions do essentially the same thing. This makes the situation more complex, conceptually, than it actually is. Although the underlying concept of each function is the same, you still have three names to remember. This situation does not occur in Java, because each absolute value method can use the same name. Indeed, Java’s standard class library includes an absolute value method, called **abs()**. This method is overloaded by Java’s **Math** class to handle all numeric types. Java determines which version of **abs()** to call based upon the type of argument.

The value of overloading is that it allows related methods to be accessed by use of a common name. Thus, the name **abs** represents the *general action* that is being performed. It is left to the compiler to choose the right *specific* version for a particular circumstance. You, the programmer, need only remember the general operation being performed. Through the application of polymorphism, several names have been reduced to one. Although this example is fairly simple, if you expand the concept, you can see how overloading can help you manage greater complexity.

When you overload a method, each version of that method can perform any activity you desire. There is no rule stating that overloaded methods must relate to one another. However, from a stylistic point of view, method overloading implies a relationship. Thus, while you can use the same name to overload unrelated methods, you should not. For example, you could use the name **sqr** to create methods that return the *square* of an integer and the *square root* of a floating-point value. But these two operations are fundamentally different. Applying method overloading in this manner defeats its original purpose. In practice, you should only overload closely related operations.

Overloading Constructors

In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception. To understand why, let's return to the **Box** class developed in the preceding chapter. Following is the latest version of **Box**:

```
class Box {
    double width;
    double height;
    double depth;

    // This is the constructor for Box.
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}
```

As you can see, the **Box()** constructor requires three parameters. This means that all declarations of **Box** objects must pass three arguments to the **Box()** constructor. For example, the following statement is currently invalid:

```
Box ob = new Box();
```

Since **Box()** requires three arguments, it's an error to call it without them. This raises some important questions. What if you simply wanted a box and did not care (or know) what its initial dimensions were? Or, what if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions? As the **Box** class is currently written, these other options are not available to you.

Fortunately, the solution to these problems is quite easy: simply overload the **Box** constructor so that it handles the situations just described. Here is a program that contains an improved version of **Box** that does just that:

```
/* Here, Box defines three constructors to initialize
   the dimensions of a box various ways.
*/
class Box {
    double width;
    double height;
    double depth;

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class OverloadCons {
    public static void main(String args[]) {
        // create boxes using the various constructors
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);

        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
    }
}
```

```

    // get volume of cube
    vol = mycube.volume();
    System.out.println("Volume of mycube is " + vol);
}
}

```

The output produced by this program is shown here:

```

Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0

```

As you can see, the proper overloaded constructor is called based upon the parameters specified when **new** is executed.

Using Objects as Parameters

So far, we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods. For example, consider the following short program:

```

// Objects may be passed to methods.
class Test {
    int a, b;

    Test(int i, int j) {
        a = i;
        b = j;
    }

    // return true if o is equal to the invoking object
    boolean equalTo(Test o) {
        if(o.a == a && o.b == b) return true;
        else return false;
    }
}

class PassOb {
    public static void main(String args[]) {
        Test ob1 = new Test(100, 22);
        Test ob2 = new Test(100, 22);
        Test ob3 = new Test(-1, -1);

        System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
        System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
    }
}

```

This program generates the following output:

```

ob1 == ob2: true
ob1 == ob3: false

```

As you can see, the `equalTo()` method inside `Test` compares two objects for equality and returns the result. That is, it compares the invoking object with the one that it is passed. If they contain the same values, then the method returns `true`. Otherwise, it returns `false`. Notice that the parameter `o` in `equalTo()` specifies `Test` as its type. Although `Test` is a class type created by the program, it is used in just the same way as Java's built-in types.

One of the most common uses of object parameters involves constructors. Frequently, you will want to construct a new object so that it is initially the same as some existing object. To do this, you must define a constructor that takes an object of its class as a parameter. For example, the following version of `Box` allows one object to initialize another:

```
// Here, Box allows one object to initialize another.

class Box {
    double width;
    double height;
    double depth;

    // Notice this constructor. It takes an object of type Box.
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class OverloadCons2 {
    public static void main(String args[]) {
        // create boxes using the various constructors
    }
}
```

```

Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);

Box myclone = new Box(mybox1); // create copy of mybox1

double vol;

// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);

// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);

// get volume of cube
vol = mycube.volume();
System.out.println("Volume of cube is " + vol);

// get volume of clone
vol = myclone.volume();
System.out.println("Volume of clone is " + vol);
}
}

```

As you will see when you begin to create your own classes, providing many forms of constructors is usually required to allow objects to be constructed in a convenient and efficient manner.

A Closer Look at Argument Passing

In general, there are two ways that a computer language can pass an argument to a subroutine. The first way is *call-by-value*. This approach copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument. The second way an argument can be passed is *call-by-reference*. In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine. As you will see, although Java uses call-by-value to pass all arguments, the precise effect differs between whether a primitive type or a reference type is passed.

When you pass a primitive type to a method, it is passed by value. Thus, a copy of the argument is made, and what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

```

// Primitive types are passed by value.
class Test {
    void meth(int i, int j) {
        i *= 2;
        j /= 2;
    }
}

```

```
class CallByValue {  
    public static void main(String args[]) {  
        Test ob = new Test();  
  
        int a = 15, b = 20;  
  
        System.out.println("a and b before call: " +  
                           a + " " + b);  
  
        ob.meth(a, b);  
  
        System.out.println("a and b after call: " +  
                           a + " " + b);  
    }  
}
```

The output from this program is shown here:

```
a and b before call: 15 20  
a and b after call: 15 20
```

As you can see, the operations that occur inside **meth()** have no effect on the values of **a** and **b** used in the call; their values here did not change to 30 and 10.

When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference. Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects act as if they are passed to methods by use of call-by-reference. Changes to the object inside the method *do* affect the object used as an argument. For example, consider the following program:

```
// Objects are passed through their references.  
  
class Test {  
    int a, b;  
  
    Test(int i, int j) {  
        a = i;  
        b = j;  
    }  
  
    // pass an object  
    void meth(Test o) {  
        o.a *= 2;  
        o.b /= 2;  
    }  
}  
  
class PassObjRef {  
    public static void main(String args[]) {  
        Test ob = new Test(15, 20);  
    }  
}
```

```

        System.out.println("ob.a and ob.b before call: " +
                           ob.a + " " + ob.b);

        ob.meth(ob);

        System.out.println("ob.a and ob.b after call: " +
                           ob.a + " " + ob.b);
    }
}

```

This program generates the following output:

```

ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10

```

As you can see, in this case, the actions inside **meth()** have affected the object used as an argument.

REMEMBER When an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does.

Returning Objects

A method can return any type of data, including class types that you create. For example, in the following program, the **incrByTen()** method returns an object in which the value of **a** is ten greater than it is in the invoking object.

```

// Returning an object.
class Test {
    int a;

    Test(int i) {
        a = i;
    }

    Test incrByTen() {
        Test temp = new Test(a+10);
        return temp;
    }
}

class RetOb {
    public static void main(String args[]) {
        Test ob1 = new Test(2);
        Test ob2;

        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);
    }
}

```

```
    ob2 = ob2.incrByTen();
    System.out.println("ob2.a after second increase: "
                       + ob2.a);
}
}
```

The output generated by this program is shown here:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

As you can see, each time **incrByTen()** is invoked, a new object is created, and a reference to it is returned to the calling routine.

The preceding program makes another important point: Since all objects are dynamically allocated using **new**, you don't need to worry about an object going out-of-scope because the method in which it was created terminates. The object will continue to exist as long as there is a reference to it somewhere in your program. When there are no references to it, the object will be reclaimed the next time garbage collection takes place.

Recursion

Java supports *recursion*. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be *recursive*.

The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N . For example, 3 factorial is $1 \times 2 \times 3 \times$, or 6. Here is how a factorial can be computed by use of a recursive method:

```
// A simple example of recursion.
class Factorial {
    // this is a recursive method
    int fact(int n) {
        int result;

        if(n==1) return 1;
        result = fact(n-1) * n;
        return result;
    }
}

class Recursion {
    public static void main(String args[]) {
        Factorial f = new Factorial();

        System.out.println("Factorial of 3 is " + f.fact(3));
        System.out.println("Factorial of 4 is " + f.fact(4));
        System.out.println("Factorial of 5 is " + f.fact(5));
    }
}
```

The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

If you are unfamiliar with recursive methods, then the operation of `fact()` may seem a bit confusing. Here is how it works. When `fact()` is called with an argument of 1, the function returns 1; otherwise, it returns the product of `fact(n-1)*n`. To evaluate this expression, `fact()` is called with `n-1`. This process repeats until `n` equals 1 and the calls to the method begin returning.

To better understand how the `fact()` method works, let's go through a short example. When you compute the factorial of 3, the first call to `fact()` will cause a second call to be made with an argument of 2. This invocation will cause `fact()` to be called a third time with an argument of 1. This call will return 1, which is then multiplied by 2 (the value of `n` in the second invocation). This result (which is 2) is then returned to the original invocation of `fact()` and multiplied by 3 (the original value of `n`). This yields the answer, 6. You might find it interesting to insert `println()` statements into `fact()`, which will show at what level each call is and what the intermediate answers are.

When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method. Recursive methods could be said to “telescope” out and back.

Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional method calls. Many recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted. If this occurs, the Java run-time system will cause an exception. However, you probably will not have to worry about this unless a recursive routine runs wild.

The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives. For example, the QuickSort sorting algorithm is quite difficult to implement in an iterative way. Also, some types of AI-related algorithms are most easily implemented using recursive solutions.

When writing recursive methods, you must have an `if` statement somewhere to force the method to return without the recursive call being executed. If you don't do this, once you call the method, it will never return. This is a very common error in working with recursion. Use `println()` statements liberally during development so that you can watch what is going on and abort execution if you see that you have made a mistake.

Here is one more example of recursion. The recursive method `printArray()` prints the first `i` elements in the array `values`.

```
// Another example that uses recursion.

class RecTest {
    int values[];
```

```
RecTest(int i) {
    values = new int[i];
}

// display array -- recursively
void printArray(int i) {
    if(i==0) return;
    else printArray(i-1);
    System.out.println("[" + (i-1) + "] " + values[i-1]);
}
}

class Recursion2 {
    public static void main(String args[]) {
        RecTest ob = new RecTest(10);
        int i;

        for(i=0; i<10; i++) ob.values[i] = i;

        ob.printArray(10);
    }
}
```

This program generates the following output:

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9
```

Introducing Access Control

As you know, encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: *access control*. Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse. For example, allowing access to data only through a well-defined set of methods, you can prevent the misuse of that data. Thus, when correctly implemented, a class creates a “black box” which may be used, but the inner workings of which are not open to tampering. However, the classes that were presented earlier do not completely meet this goal. For example, consider the **Stack** class shown at the end of Chapter 6. While it is true that the methods **push()** and **pop()** do provide a controlled interface to the stack, this interface is not enforced. That is, it is possible for another part of the program to bypass these methods and access the stack directly. Of course, in the wrong hands, this could lead to trouble. In this section, you will be introduced to the mechanism by which you can precisely control access to the various members of a class.

How a member can be accessed is determined by the *access modifier* attached to its declaration. Java supplies a rich set of access modifiers. Some aspects of access control are related mostly to inheritance or packages. (A *package* is, essentially, a grouping of classes.) These parts of Java's access control mechanism will be discussed later. Here, let's begin by examining access control as it applies to a single class. Once you understand the fundamentals of access control, the rest will be easy.

Java's access modifiers are **public**, **private**, and **protected**. Java also defines a default access level. **protected** applies only when inheritance is involved. The other access modifiers are described next.

Let's begin by defining **public** and **private**. When a member of a class is modified by **public**, then that member can be accessed by any other code. When a member of a class is specified as **private**, then that member can only be accessed by other members of its class. Now you can understand why **main()** has always been preceded by the **public** modifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package. (Packages are discussed in the following chapter.)

In the classes developed so far, all members of a class have used the default access mode. However, this is not what you will typically want to be the case. Usually, you will want to restrict access to the data members of a class—allowing access only through methods. Also, there will be times when you will want to define methods that are private to a class.

An access modifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here is an example:

```
public int i;  
private double j;  
  
private int myMethod(int a, char b) { //...
```

To understand the effects of public and private access, consider the following program:

```
/* This program demonstrates the difference between  
   public and private.  
*/  
class Test {  
    int a; // default access  
    public int b; // public access  
    private int c; // private access  
  
    // methods to access c  
    void setc(int i) { // set c's value  
        c = i;  
    }  
    int getc() { // get c's value  
        return c;  
    }  
}
```

```
class AccessTest {  
    public static void main(String args[]) {  
        Test ob = new Test();  
  
        // These are OK, a and b may be accessed directly  
        ob.a = 10;  
        ob.b = 20;  
  
        // This is not OK and will cause an error  
        // ob.c = 100; // Error!  
  
        // You must access c through its methods  
        ob.setc(100); // OK  
        System.out.println("a, b, and c: " + ob.a + " " +  
                           ob.b + " " + ob.getc());  
    }  
}
```

As you can see, inside the **Test** class, **a** uses default access, which for this example is the same as specifying **public**. **b** is explicitly specified as **public**. Member **c** is given private access. This means that it cannot be accessed by code outside of its class. So, inside the **AccessTest** class, **c** cannot be used directly. It must be accessed through its public methods: **setc()** and **getc()**. If you were to remove the comment symbol from the beginning of the following line,

```
// ob.c = 100; // Error!
```

then you would not be able to compile this program because of the access violation.

To see how access control can be applied to a more practical example, consider the following improved version of the **Stack** class shown at the end of Chapter 6.

```
// This class defines an integer stack that can hold 10 values.  
class Stack {  
    /* Now, both stck and tos are private. This means  
       that they cannot be accidentally or maliciously  
       altered in a way that would be harmful to the stack.  
    */  
    private int stck[] = new int[10];  
    private int tos;  
  
    // Initialize top-of-stack  
    Stack() {  
        tos = -1;  
    }  
  
    // Push an item onto the stack  
    void push(int item) {  
        if(tos==9)  
            System.out.println("Stack is full.");  
        else  
            stck[++tos] = item;  
    }  
}
```

```

    // Pop an item from the stack
    int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}

```

As you can see, now both **stck**, which holds the stack, and **tos**, which is the index of the top of the stack, are specified as **private**. This means that they cannot be accessed or altered except through **push()** and **pop()**. Making **tos** private, for example, prevents other parts of your program from inadvertently setting it to a value that is beyond the end of the **stck** array.

The following program demonstrates the improved **Stack** class. Try removing the commented-out lines to prove to yourself that the **stck** and **tos** members are, indeed, inaccessible.

```

class TestStack {
    public static void main(String args[]) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();

        // push some numbers onto the stack
        for(int i=0; i<10; i++) mystack1.push(i);
        for(int i=10; i<20; i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<10; i++)
            System.out.println(mystack1.pop());

        System.out.println("Stack in mystack2:");

        for(int i=0; i<10; i++)
            System.out.println(mystack2.pop());

        // these statements are not legal
        // mystack1.tos = -2;
        // mystack2.stck[3] = 100;
    }
}

```

Although methods will usually provide access to the data defined by a class, this does not always have to be the case. It is perfectly proper to allow an instance variable to be public when there is good reason to do so. For example, most of the simple classes in this book were created with little concern about controlling access to instance variables for the sake of simplicity. However, in most real-world classes, you will need to allow operations on data only through methods. The next chapter will return to the topic of access control. As you will see, it is particularly important when inheritance is involved.

Understanding static

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be **static**. The most common example of a **static** member is **main()**. **main()** is declared as **static** because it must be called before any objects exist.

Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

Methods declared as **static** have several restrictions:

- They can only directly call other **static** methods.
- They can only directly access **static** data.
- They cannot refer to **this** or **super** in any way. (The keyword **super** relates to inheritance and is described in the next chapter.)

If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block:

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
    static int a = 3;
    static int b;

    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }

    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }

    public static void main(String args[]) {
        meth(42);
    }
}
```

As soon as the **UseStatic** class is loaded, all of the **static** statements are run. First, **a** is set to **3**, then the **static** block executes, which prints a message and then initializes **b** to **a*4** or **12**. Then **main()** is called, which calls **meth()**, passing **42** to **x**. The three **println()** statements refer to the two **static** variables **a** and **b**, as well as to the local variable **x**.

Here is the output of the program:

```
Static block initialized.
x = 42
a = 3
b = 12
```

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. For example, if you wish to call a **static** method from outside its class, you can do so using the following general form:

```
classname.method( )
```

Here, *classname* is the name of the class in which the **static** method is declared. As you can see, this format is similar to that used to call non-**static** methods through object-reference variables. A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.

Here is an example. Inside **main()**, the **static** method **callme()** and the **static** variable **b** are accessed through their class name **StaticDemo**.

```
class StaticDemo {
    static int a = 42;
    static int b = 99;

    static void callme() {
        System.out.println("a = " + a);
    }
}

class StaticByName {
    public static void main(String args[]) {
        StaticDemo.callme();
        System.out.println("b = " + StaticDemo.b);
    }
}
```

Here is the output of this program:

```
a = 42
b = 99
```

Introducing **final**

A field can be declared as **final**. Doing so prevents its contents from being modified, making it, essentially, a constant. This means that you must initialize a **final** field when it is declared. You can do this in one of two ways: First, you can give it a value when it is declared. Second, you can assign it a value within a constructor. The first approach is the most common. Here is an example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Subsequent parts of your program can now use **FILE_OPEN**, etc., as if they were constants, without fear that a value has been changed. It is a common coding convention to choose all uppercase identifiers for **final** fields, as this example shows.

In addition to fields, both method parameters and local variables can be declared **final**. Declaring a parameter **final** prevents it from being changed within the method. Declaring a local variable **final** prevents it from being assigned a value more than once.

The keyword **final** can also be applied to methods, but its meaning is substantially different than when it is applied to variables. This additional usage of **final** is described in the next chapter, when inheritance is described.

Arrays Revisited

Arrays were introduced earlier in this book, before classes had been discussed. Now that you know about classes, an important point can be made about arrays: they are implemented as objects. Because of this, there is a special array attribute that you will want to take advantage of. Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its **length** instance variable. All arrays have this variable, and it will always hold the size of the array. Here is a program that demonstrates this property:

```
// This program demonstrates the length array member.
class Length {
    public static void main(String args[]) {
        int a1[] = new int[10];
        int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
        int a3[] = {4, 3, 2, 1};

        System.out.println("length of a1 is " + a1.length);
        System.out.println("length of a2 is " + a2.length);
        System.out.println("length of a3 is " + a3.length);
    }
}
```

This program displays the following output:

```
length of a1 is 10
length of a2 is 8
length of a3 is 4
```

As you can see, the size of each array is displayed. Keep in mind that the value of **length** has nothing to do with the number of elements that are actually in use. It only reflects the number of elements that the array is designed to hold.

You can put the **length** member to good use in many situations. For example, here is an improved version of the **Stack** class. As you might recall, the earlier versions of this class

always created a ten-element stack. The following version lets you create stacks of any size. The value of `stck.length` is used to prevent the stack from overflowing.

```
// Improved Stack class that uses the length array member.
class Stack {
    private int stck[];
    private int tos;

    // allocate and initialize stack
    Stack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // Push an item onto the stack
    void push(int item) {
        if(tos==stck.length-1) // use length member
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }

    // Pop an item from the stack
    int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}

class TestStack2 {
    public static void main(String args[]) {
        Stack mystack1 = new Stack(5);
        Stack mystack2 = new Stack(8);

        // push some numbers onto the stack
        for(int i=0; i<5; i++) mystack1.push(i);
        for(int i=0; i<8; i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());

        System.out.println("Stack in mystack2:");
        for(int i=0; i<8; i++)
            System.out.println(mystack2.pop());
    }
}
```

Notice that the program creates two stacks: one five elements deep and the other eight elements deep. As you can see, the fact that arrays maintain their own length information makes it easy to create stacks of any size.

Introducing Nested and Inner Classes

It is possible to define a class within another class; such classes are known as *nested classes*. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.

There are two types of nested classes: *static* and *non-static*. A static nested class is one that has the **static** modifier applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

The most important type of nested class is the *inner* class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

The following program illustrates how to define and use an inner class. The class named **Outer** has one instance variable named **outer_x**, one instance method named **test()**, and defines one inner class called **Inner**.

```
// Demonstrate an inner class.
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // this is an inner class
    class Inner {
        void display() {
            System.out.println("display: outer_x = " + outer_x);
        }
    }
}

class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Output from this application is shown here:

```
display: outer_x = 100
```

In the program, an inner class named **Inner** is defined within the scope of class **Outer**. Therefore, any code in class **Inner** can directly access the variable **outer_x**. An instance method named **display()** is defined inside **Inner**. This method displays **outer_x** on the standard output stream. The **main()** method of **InnerClassDemo** creates an instance of class **Outer** and invokes its **test()** method. That method creates an instance of class **Inner** and the **display()** method is called.

It is important to realize that an instance of **Inner** can be created only in the context of class **Outer**. The Java compiler generates an error message otherwise. In general, an inner class instance is often created by code within its enclosing scope, as the example does.

As explained, an inner class has access to all of the members of its enclosing class, but the reverse is not true. Members of the inner class are known only within the scope of the inner class and may not be used by the outer class. For example,

```
// This program will not compile.
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // this is an inner class
    class Inner {
        int y = 10; // y is local to Inner

        void display() {
            System.out.println("display: outer_x = " + outer_x);
        }
    }

    void showy() {
        System.out.println(y); // error, y not known here!
    }
}

class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Here, **y** is declared as an instance variable of **Inner**. Thus, it is not known outside of that class and it cannot be used by **showy()**.

Although we have been focusing on inner classes declared as members within an outer class scope, it is possible to define inner classes within any block scope. For example, you can define a nested class within the block defined by a method or even within the body of a **for** loop, as this next program shows:

```
// Define an inner class within a for loop.
class Outer {
    int outer_x = 100;

    void test() {
        for(int i=0; i<10; i++) {
            class Inner {
                void display() {
                    System.out.println("display: outer_x = " + outer_x);
                }
            }
            Inner inner = new Inner();
            inner.display();
        }
    }
}

class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

The output from this version of the program is shown here:

```
display: outer_x = 100
```

While nested classes are not applicable to all situations, they are particularly helpful when handling events. We will return to the topic of nested classes in Chapter 24. There you will see how inner classes can be used to simplify the code needed to handle certain types of events. You will also learn about *anonymous inner classes*, which are inner classes that don't have a name.

One final point: Nested classes were not allowed by the original 1.0 specification for Java. They were added by Java 1.1.

Exploring the String Class

Although the **String** class will be examined in depth in Part II of this book, a short exploration of it is warranted now, because we will be using strings in some of the example programs shown toward the end of Part I. **String** is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming.

The first thing to understand about strings is that every string you create is actually an object of type **String**. Even string constants are actually **String** objects. For example, in the statement

```
System.out.println("This is a String, too");
```

the string "This is a String, too" is a **String** object.

The second thing to understand about strings is that objects of type **String** are immutable; once a **String** object is created, its contents cannot be altered. While this may seem like a serious restriction, it is not, for two reasons:

- If you need to change a string, you can always create a new one that contains the modifications.
- Java defines peer classes of **String**, called **StringBuffer** and **StringBuilder**, which allow strings to be altered, so all of the normal string manipulations are still available in Java. (**StringBuffer** and **StringBuilder** are described in Part II of this book.)

Strings can be constructed in a variety of ways. The easiest is to use a statement like this:

```
String myString = "this is a test";
```

Once you have created a **String** object, you can use it anywhere that a string is allowed. For example, this statement displays **myString**:

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

Java defines one operator for **String** objects: **+**. It is used to concatenate two strings. For example, this statement

```
String myString = "I" + " like " + "Java.;"
```

results in **myString** containing "I like Java."

The following program demonstrates the preceding concepts:

```
// Demonstrating Strings.
class StringDemo {
    public static void main(String args[]) {
        String strOb1 = "First String";
        String strOb2 = "Second String";
        String strOb3 = strOb1 + " and " + strOb2;

        System.out.println(strOb1);
```

```
        System.out.println(strOb2);
        System.out.println(strOb3);
    }
}
```

The output produced by this program is shown here:

```
First String
Second String
First String and Second String
```

The **String** class contains several methods that you can use. Here are a few. You can test two strings for equality by using **equals()**. You can obtain the length of a string by calling the **length()** method. You can obtain the character at a specified index within a string by calling **charAt()**. The general forms of these three methods are shown here:

```
boolean equals(secondStr)
int length()
char charAt(index)
```

Here is a program that demonstrates these methods:

```
// Demonstrating some String methods.
class StringDemo2 {
    public static void main(String args[]) {
        String strOb1 = "First String";
        String strOb2 = "Second String";
        String strOb3 = strOb1;

        System.out.println("Length of strOb1: " +
                           strOb1.length());

        System.out.println("Char at index 3 in strOb1: " +
                           strOb1.charAt(3));

        if(strOb1.equals(strOb2))
            System.out.println("strOb1 == strOb2");
        else
            System.out.println("strOb1 != strOb2");

        if(strOb1.equals(strOb3))
            System.out.println("strOb1 == strOb3");
        else
            System.out.println("strOb1 != strOb3");
    }
}
```

This program generates the following output:

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

Of course, you can have arrays of strings, just like you can have arrays of any other type of object. For example:

```
// Demonstrate String arrays.
class StringDemo3 {
    public static void main(String args[]) {
        String str[] = { "one", "two", "three" };

        for(int i=0; i<str.length; i++)
            System.out.println("str[" + i + "] : " +
                               str[i]);
    }
}
```

Here is the output from this program:

```
str[0]: one
str[1]: two
str[2]: three
```

As you will see in the following section, string arrays play an important part in many Java programs.

Using Command-Line Arguments

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing *command-line arguments* to `main()`. A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy—they are stored as strings in a `String` array passed to the `args` parameter of `main()`. The first command-line argument is stored at `args[0]`, the second at `args[1]`, and so on. For example, the following program displays all of the command-line arguments that it is called with:

```
// Display all command-line arguments.
class CommandLine {
    public static void main(String args[]) {
        for(int i=0; i<args.length; i++)
            System.out.println("args[" + i + "] : " +
                               args[i]);
    }
}
```

Try executing this program, as shown here:

```
java CommandLine this is a test 100 -1
```

When you do, you will see the following output:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

REMEMBER All command-line arguments are passed as strings. You must convert numeric values to their internal forms manually, as explained in Chapter 17.

Varargs: Variable-Length Arguments

Beginning with JDK 5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called *varargs* and it is short for *variable-length arguments*. A method that takes a variable number of arguments is called a *variable-arity method*, or simply a *varargs method*.

Situations that require that a variable number of arguments be passed to a method are not unusual. For example, a method that opens an Internet connection might take a user name, password, filename, protocol, and so on, but supply defaults if some of this information is not provided. In this situation, it would be convenient to pass only the arguments to which the defaults did not apply. Another example is the `printf()` method that is part of Java's I/O library. As you will see in Chapter 20, it takes a variable number of arguments, which it formats and then outputs.

Prior to JDK 5, variable-length arguments could be handled two ways, neither of which was particularly pleasing. First, if the maximum number of arguments was small and known, then you could create overloaded versions of the method, one for each way the method could be called. Although this works and is suitable for some cases, it applies to only a narrow class of situations.

In cases where the maximum number of potential arguments was larger, or unknowable, a second approach was used in which the arguments were put into an array, and then the array was passed to the method. This approach is illustrated by the following program:

```
// Use an array to pass a variable number of
// arguments to a method. This is the old-style
// approach to variable-length arguments.
class PassArray {
    static void vaTest(int v[]) {
        System.out.print("Number of args: " + v.length +
                         " Contents: ");
        for(int x : v)
            System.out.print(x + " ");
        System.out.println();
    }

    public static void main(String args[])
    {
        // Notice how an array must be created to
        // hold the arguments.
        int n1[] = { 10 };
        int n2[] = { 1, 2, 3 };
        int n3[] = { };

        vaTest(n1); // 1 arg
        vaTest(n2); // 3 args
        vaTest(n3); // no args
    }
}
```

The output from the program is shown here:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

In the program, the method `vaTest()` is passed its arguments through the array `v`. This old-style approach to variable-length arguments does enable `vaTest()` to take an arbitrary number of arguments. However, it requires that these arguments be manually packaged into an array prior to calling `vaTest()`. Not only is it tedious to construct an array each time `vaTest()` is called, it is potentially error-prone. The varargs feature offers a simpler, better option.

A variable-length argument is specified by three periods (...). For example, here is how `vaTest()` is written using a vararg:

```
static void vaTest(int ... v) {
```

This syntax tells the compiler that `vaTest()` can be called with zero or more arguments. As a result, `v` is implicitly declared as an array of type `int[]`. Thus, inside `vaTest()`, `v` is accessed using the normal array syntax. Here is the preceding program rewritten using a vararg:

```
// Demonstrate variable-length arguments.
class VarArgs {

    // vaTest() now uses a vararg.
    static void vaTest(int ... v) {
        System.out.print("Number of args: " + v.length +
                        " Contents: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    public static void main(String args[])
    {
        // Notice how vaTest() can be called with a
        // variable number of arguments.
        vaTest(10);           // 1 arg
        vaTest(1, 2, 3);     // 3 args
        vaTest();             // no args
    }
}
```

The output from the program is the same as the original version.

There are two important things to notice about this program. First, as explained, inside `vaTest()`, `v` is operated on as an array. This is because `v` is an array. The ... syntax simply tells the compiler that a variable number of arguments will be used, and that these arguments will be stored in the array referred to by `v`. Second, in `main()`, `vaTest()` is called with different numbers of arguments, including no arguments at all. The arguments are automatically put in an array and passed to `v`. In the case of no arguments, the length of the array is zero.

A method can have “normal” parameters along with a variable-length parameter. However, the variable-length parameter must be the last parameter declared by the method. For example, this method declaration is perfectly acceptable:

```
int doIt(int a, int b, double c, int ... vals) {
```

In this case, the first three arguments used in a call to `doIt()` are matched to the first three parameters. Then, any remaining arguments are assumed to belong to `vals`.

Remember, the varargs parameter must be last. For example, the following declaration is incorrect:

```
int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Error!
```

Here, there is an attempt to declare a regular parameter after the varargs parameter, which is illegal.

There is one more restriction to be aware of: there must be only one varargs parameter. For example, this declaration is also invalid:

```
int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Error!
```

The attempt to declare the second varargs parameter is illegal.

Here is a reworked version of the `vaTest()` method that takes a regular argument and a variable-length argument:

```
// Use varargs with standard arguments.
class VarArgs2 {

    // Here, msg is a normal parameter and v is a
    // varargs parameter.
    static void vaTest(String msg, int ... v) {
        System.out.print(msg + v.length +
                          " Contents: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    public static void main(String args[])
    {
        vaTest("One vararg: ", 10);
        vaTest("Three varargs: ", 1, 2, 3);
        vaTest("No varargs: ");
    }
}
```

The output from this program is shown here:

```
One vararg: 1 Contents: 10
Three varargs: 3 Contents: 1 2 3
No varargs: 0 Contents:
```

Overloading Vararg Methods

You can overload a method that takes a variable-length argument. For example, the following program overloads `vaTest()` three times:

```
// Varargs and overloading.
class VarArgs3 {

    static void vaTest(int ... v) {
        System.out.print("vaTest(int ...): " +
                         "Number of args: " + v.length +
                         " Contents: ");

        for(int x : v)
            System.out.print(x + " ");
        System.out.println();
    }

    static void vaTest(boolean ... v) {
        System.out.print("vaTest(boolean ...): " +
                         "Number of args: " + v.length +
                         " Contents: ");

        for(boolean x : v)
            System.out.print(x + " ");
        System.out.println();
    }

    static void vaTest(String msg, int ... v) {
        System.out.print("vaTest(String, int ...): " +
                         msg + v.length +
                         " Contents: ");

        for(int x : v)
            System.out.print(x + " ");
        System.out.println();
    }

    public static void main(String args[])
    {
        vaTest(1, 2, 3);
        vaTest("Testing: ", 10, 20);
        vaTest(true, false, false);
    }
}
```

The output produced by this program is shown here:

```
vaTest(int ...): Number of args: 3 Contents: 1 2 3
vaTest(String, int ...): Testing: 2 Contents: 10 20
vaTest(boolean ...): Number of args: 3 Contents: true false false
```

This program illustrates both ways that a varargs method can be overloaded. First, the types of its vararg parameter can differ. This is the case for `vaTest(int ...)` and `vaTest(boolean ...)`. Remember, the `...` causes the parameter to be treated as an array of the specified type. Therefore, just as you can overload methods by using different types of array parameters, you can overload vararg methods by using different types of varargs. In this case, Java uses the type difference to determine which overloaded method to call.

The second way to overload a varargs method is to add one or more normal parameters. This is what was done with `vaTest(String, int ...)`. In this case, Java uses both the number of arguments and the type of the arguments to determine which method to call.

NOTE A varargs method can also be overloaded by a non-varargs method. For example, `vaTest(int x)` is a valid overload of `vaTest()` in the foregoing program. This version is invoked only when one `int` argument is present. When two or more `int` arguments are passed, the varargs version `vaTest (int...v)` is used.

Varargs and Ambiguity

Somewhat unexpected errors can result when overloading a method that takes a variable-length argument. These errors involve ambiguity because it is possible to create an ambiguous call to an overloaded varargs method. For example, consider the following program:

```
// Varargs, overloading, and ambiguity.
//
// This program contains an error and will
// not compile!
class VarArgs4 {

    static void vaTest(int ... v) {
        System.out.print("vaTest(int ...): " +
                        "Number of args: " + v.length +
                        " Contents: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    static void vaTest(boolean ... v) {
        System.out.print("vaTest(boolean ...): " +
                        "Number of args: " + v.length +
                        " Contents: ");

        for(boolean x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    public static void main(String args[])
    {
```

```
    vaTest(1, 2, 3); // OK
    vaTest(true, false, false); // OK

    vaTest(); // Error: Ambiguous!
}
}
```

In this program, the overloading of **vaTest()** is perfectly correct. However, this program will not compile because of the following call:

```
vaTest(); // Error: Ambiguous!
```

Because the vararg parameter can be empty, this call could be translated into a call to **vaTest(int ...)** or **vaTest(boolean ...)**. Both are equally valid. Thus, the call is inherently ambiguous.

Here is another example of ambiguity. The following overloaded versions of **vaTest()** are inherently ambiguous even though one takes a normal parameter:

```
static void vaTest(int ... v) { // ...
static void vaTest(int n, int ... v) { // ...

```

Although the parameter lists of **vaTest()** differ, there is no way for the compiler to resolve the following call:

```
vaTest(1)
```

Does this translate into a call to **vaTest(int ...)**, with one varargs argument, or into a call to **vaTest(int, int ...)** with no varargs arguments? There is no way for the compiler to answer this question. Thus, the situation is ambiguous.

Because of ambiguity errors like those just shown, sometimes you will need to forego overloading and simply use two different method names. Also, in some cases, ambiguity errors expose a conceptual flaw in your code, which you can remedy by more carefully crafting a solution.

CHAPTER

8

Inheritance

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a *superclass*. The class that does the inheriting is called a *subclass*. Therefore, a subclass is a specialized version of a superclass. It inherits all of the members defined by the superclass and adds its own, unique elements.

Inheritance Basics

To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword. To see how, let's begin with a short example. The following program creates a superclass called **A** and a subclass called **B**. Notice how the keyword **extends** is used to create a subclass of **A**.

```
// A simple example of inheritance.

// Create a superclass.
class A {
    int i, j;

    void showij() {
        System.out.println("i and j: " + i + " " + j);
    }
}

// Create a subclass by extending class A.
class B extends A {
    int k;

    void showk() {
        System.out.println("k: " + k);
    }
}
```

```

void sum() {
    System.out.println("i+j+k: " + (i+j+k));
}
}

class SimpleInheritance {
    public static void main(String args []) {
        A superOb = new A();
        B subOb = new B();

        // The superclass may be used by itself.
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Contents of superOb: ");
        superOb.showij();
        System.out.println();

        /* The subclass has access to all public members of
           its superclass. */
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println("Contents of subOb: ");
        subOb.showij();
        subOb.showk();
        System.out.println();

        System.out.println("Sum of i, j and k in subOb:");
        subOb.sum();
    }
}
}

```

The output from this program is shown here:

```

Contents of superOb:
i and j: 10 20

Contents of subOb:
i and j: 7 8
k: 9

Sum of i, j and k in subOb:
i+j+k: 24

```

As you can see, the subclass **B** includes all of the members of its superclass, **A**. This is why **subOb** can access **i** and **j** and call **showij()**. Also, inside **sum()**, **i** and **j** can be referred to directly, as if they were part of **B**.

Even though **A** is a superclass for **B**, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a superclass for another subclass.

The general form of a **class** declaration that inherits a superclass is shown here:

```
class subclass-name extends superclass-name {  
    // body of class  
}
```

You can only specify one superclass for any subclass that you create. Java does not support the inheritance of multiple superclasses into a single subclass. You can, as stated, create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**. For example, consider the following simple class hierarchy:

```
/* In a class hierarchy, private members remain  
private to their class.  
  
This program contains an error and will not  
compile.  
*/  
  
// Create a superclass.  
class A {  
    int i; // public by default  
    private int j; // private to A  
  
    void setij(int x, int y) {  
        i = x;  
        j = y;  
    }  
}  
  
// A's j is not accessible here.  
class B extends A {  
    int total;  
  
    void sum() {  
        total = i + j; // ERROR, j is not accessible here  
    }  
}  
  
class Access {  
    public static void main(String args[]) {  
        B subOb = new B();  
  
        subOb.setij(10, 12);  
  
        subOb.sum();  
        System.out.println("Total is " + subOb.total);  
    }  
}
```

This program will not compile because the use of **j** inside the **sum()** method of **B** causes an access violation. Since **j** is declared as **private**, it is only accessible by other members of its own class. Subclasses have no access to it.

REMEMBER A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

A More Practical Example

Let's look at a more practical example that will help illustrate the power of inheritance. Here, the final version of the **Box** class developed in the preceding chapter will be extended to include a fourth component called **weight**. Thus, the new class will contain a box's width, height, depth, and weight.

```
// This program uses inheritance to extend Box.
class Box {
    double width;
    double height;
    double depth;

    // construct clone of an object
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

// Here, Box is extended to include weight.
class BoxWeight extends Box {
```

```

        double weight; // weight of box

        // constructor for BoxWeight
        BoxWeight(double w, double h, double d, double m) {
            width = w;
            height = h;
            depth = d;
            weight = m;
        }
    }

    class DemoBoxWeight {
        public static void main(String args[]) {
            BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
            BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
            double vol;

            vol = mybox1.volume();
            System.out.println("Volume of mybox1 is " + vol);
            System.out.println("Weight of mybox1 is " + mybox1.weight);
            System.out.println();

            vol = mybox2.volume();
            System.out.println("Volume of mybox2 is " + vol);
            System.out.println("Weight of mybox2 is " + mybox2.weight);
        }
    }
}

```

The output from this program is shown here:

```

Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3

```

```

Volume of mybox2 is 24.0
Weight of mybox2 is 0.076

```

BoxWeight inherits all of the characteristics of **Box** and adds to them the **weight** component. It is not necessary for **BoxWeight** to re-create all of the features found in **Box**. It can simply extend **Box** to meet its own purposes.

A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses. Each subclass can precisely tailor its own classification. For example, the following class inherits **Box** and adds a color attribute:

```

// Here, Box is extended to include color.
class ColorBox extends Box {
    int color; // color of box

    ColorBox(double w, double h, double d, int c) {
        width = w;
        height = h;
        depth = d;
        color = c;
    }
}

```

Remember, once you have created a superclass that defines the general aspects of an object, that superclass can be inherited to form specialized classes. Each subclass simply adds its own unique attributes. This is the essence of inheritance.

A Superclass Variable Can Reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. You will find this aspect of inheritance quite useful in a variety of situations. For example, consider the following:

```
class RefDemo {
    public static void main(String args[]) {
        BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
        Box plainbox = new Box();
        double vol;

        vol = weightbox.volume();
        System.out.println("Volume of weightbox is " + vol);
        System.out.println("Weight of weightbox is " +
                           weightbox.weight);
        System.out.println();

        // assign BoxWeight reference to Box reference
        plainbox = weightbox;

        vol = plainbox.volume(); // OK, volume() defined in Box
        System.out.println("Volume of plainbox is " + vol);

        /* The following statement is invalid because plainbox
           does not define a weight member. */
        // System.out.println("Weight of plainbox is " + plainbox.weight);
    }
}
```

Here, **weightbox** is a reference to **BoxWeight** objects, and **plainbox** is a reference to **Box** objects. Since **BoxWeight** is a subclass of **Box**, it is permissible to assign **plainbox** a reference to the **weightbox** object.

It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed. That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass. This is why **plainbox** can't access **weight** even when it refers to a **BoxWeight** object. If you think about it, this makes sense, because the superclass has no knowledge of what a subclass adds to it. This is why the last line of code in the preceding fragment is commented out. It is not possible for a **Box** reference to access the **weight** field, because **Box** does not define one.

Although the preceding may seem a bit esoteric, it has some important practical applications—two of which are discussed later in this chapter.

Using super

In the preceding examples, classes derived from **Box** were not implemented as efficiently or as robustly as they could have been. For example, the constructor for **BoxWeight** explicitly initializes the **width**, **height**, and **depth** fields of **Box**. Not only does this duplicate code found in its superclass, which is inefficient, but it implies that a subclass must be granted access to these members. However, there will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own. Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.

super has two general forms. The first calls the superclass' constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass. Each use is examined here.

Using super to Call Superclass Constructors

A subclass can call a constructor defined by its superclass by use of the following form of **super**:

```
super(arg-list);
```

Here, *arg-list* specifies any arguments needed by the constructor in the superclass. **super()** must always be the first statement executed inside a subclass' constructor.

To see how **super()** is used, consider this improved version of the **BoxWeight** class:

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
    double weight; // weight of box

    // initialize width, height, and depth using super()
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // call superclass constructor
        weight = m;
    }
}
```

Here, **BoxWeight()** calls **super()** with the arguments **w**, **h**, and **d**. This causes the **Box** constructor to be called, which initializes **width**, **height**, and **depth** using these values. **BoxWeight** no longer initializes these values itself. It only needs to initialize the value unique to it: **weight**. This leaves **Box** free to make these values **private** if desired.

In the preceding example, **super()** was called with three arguments. Since constructors can be overloaded, **super()** can be called using any form defined by the superclass. The constructor executed will be the one that matches the arguments. For example, here is a complete implementation of **BoxWeight** that provides constructors for the various ways that

a box can be constructed. In each case, `super()` is called using the appropriate arguments. Notice that `width`, `height`, and `depth` have been made private within `Box`.

```
// A complete implementation of BoxWeight.
class Box {
    private double width;
    private double height;
    private double depth;

    // construct clone of an object
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
    double weight; // weight of box

    // construct clone of an object
    BoxWeight(BoxWeight ob) { // pass object to constructor
        super(ob);
        weight = ob.weight;
    }

    // constructor when all parameters are specified
    BoxWeight(double w, double h, double d, double m) {
```

```
super(w, h, d); // call superclass constructor
    weight = m;
}

// default constructor
BoxWeight() {
    super();
    weight = -1;
}

// constructor used when cube is created
BoxWeight(double len, double m) {
    super(len);
    weight = m;
}
}

class DemoSuper {
    public static void main(String args[]) {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
        BoxWeight mybox3 = new BoxWeight(); // default
        BoxWeight mycube = new BoxWeight(3, 2);
        BoxWeight myclone = new BoxWeight(mybox1);
        double vol;

        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        System.out.println("Weight of mybox1 is " + mybox1.weight);
        System.out.println();

        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        System.out.println("Weight of mybox2 is " + mybox2.weight);
        System.out.println();

        vol = mybox3.volume();
        System.out.println("Volume of mybox3 is " + vol);
        System.out.println("Weight of mybox3 is " + mybox3.weight);
        System.out.println();

        vol = myclone.volume();
        System.out.println("Volume of myclone is " + vol);
        System.out.println("Weight of myclone is " + myclone.weight);
        System.out.println();
    }
}
```

This program generates the following output:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3

Volume of mybox2 is 24.0
Weight of mybox2 is 0.076

Volume of mybox3 is -1.0
Weight of mybox3 is -1.0

Volume of myclone is 3000.0
Weight of myclone is 34.3

Volume of mycube is 27.0
Weight of mycube is 2.0
```

Pay special attention to this constructor in **BoxWeight**:

```
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
    super(ob);
    weight = ob.weight;
}
```

Notice that **super()** is passed an object of type **BoxWeight**—not of type **Box**. This still invokes the constructor **Box(Box ob)**. As mentioned earlier, a superclass variable can be used to reference any object derived from that class. Thus, we are able to pass a **BoxWeight** object to the **Box** constructor. Of course, **Box** only has knowledge of its own members.

Let's review the key concepts behind **super()**. When a subclass calls **super()**, it is calling the constructor of its immediate superclass. Thus, **super()** always refers to the superclass immediately above the calling class. This is true even in a multileveled hierarchy. Also, **super()** must always be the first statement executed inside a subclass constructor.

A Second Use for **super**

The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

super.member

Here, *member* can be either a method or an instance variable.

This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:

```
// Using super to overcome name hiding.
class A {
    int i;
}

// Create a subclass by extending class A.
```

```

class B extends A {
    int i; // this i hides the i in A

    B(int a, int b) {
        super.i = a; // i in A
        i = b; // i in B
    }

    void show() {
        System.out.println("i in superclass: " + super.i);
        System.out.println("i in subclass: " + i);
    }
}

class UseSuper {
    public static void main(String args[]) {
        B subOb = new B(1, 2);

        subOb.show();
    }
}

```

This program displays the following:

```
i in superclass: 1
i in subclass: 2
```

Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass. As you will see, **super** can also be used to call methods that are hidden by a subclass.

Creating a Multilevel Hierarchy

Up to this point, we have been using simple class hierarchies that consist of only a superclass and a subclass. However, you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another. For example, given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is a subclass of **A**. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, **C** inherits all aspects of **B** and **A**. To see how a multilevel hierarchy can be useful, consider the following program. In it, the subclass **BoxWeight** is used as a superclass to create the subclass called **Shipment**. **Shipment** inherits all of the traits of **BoxWeight** and **Box**, and adds a field called **cost**, which holds the cost of shipping such a parcel.

```

// Extend BoxWeight to include shipping costs.

// Start with Box.
class Box {
    private double width;
    private double height;
    private double depth;

```

```
// construct clone of an object
Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
}

// constructor used when all dimensions specified
Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
}

// constructor used when no dimensions specified
Box() {
    width = -1; // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1; // box
}

// constructor used when cube is created
Box(double len) {
    width = height = depth = len;
}

// compute and return volume
double volume() {
    return width * height * depth;
}
}

// Add weight.
class BoxWeight extends Box {
    double weight; // weight of box

    // construct clone of an object
    BoxWeight(BoxWeight ob) { // pass object to constructor
        super(ob);
        weight = ob.weight;
    }

    // constructor when all parameters are specified
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // call superclass constructor
        weight = m;
    }

    // default constructor
    BoxWeight() {
        super();
        weight = -1;
    }
}
```

```
// constructor used when cube is created
BoxWeight(double len, double m) {
    super(len);
    weight = m;
}
}

// Add shipping costs.
class Shipment extends BoxWeight {
    double cost;

    // construct clone of an object
    Shipment(Shipment ob) { // pass object to constructor
        super(ob);
        cost = ob.cost;
    }

    // constructor when all parameters are specified
    Shipment(double w, double h, double d,
             double m, double c) {
        super(w, h, d, m); // call superclass constructor
        cost = c;
    }

    // default constructor
    Shipment() {
        super();
        cost = -1;
    }

    // constructor used when cube is created
    Shipment(double len, double m, double c) {
        super(len, m);
        cost = c;
    }
}

class DemoShipment {
    public static void main(String args[]) {
        Shipment shipment1 =
            new Shipment(10, 20, 15, 10, 3.41);
        Shipment shipment2 =
            new Shipment(2, 3, 4, 0.76, 1.28);

        double vol;

        vol = shipment1.volume();
        System.out.println("Volume of shipment1 is " + vol);
        System.out.println("Weight of shipment1 is "
                           + shipment1.weight);
        System.out.println("Shipping cost: $" + shipment1.cost);
        System.out.println();
    }
}
```

```

        vol = shipment2.volume();
        System.out.println("Volume of shipment2 is " + vol);
        System.out.println("Weight of shipment2 is "
                           + shipment2.weight);
        System.out.println("Shipping cost: $" + shipment2.cost);
    }
}

```

The output of this program is shown here:

```

Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41

```

```

Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28

```

Because of inheritance, **Shipment** can make use of the previously defined classes of **Box** and **BoxWeight**, adding only the extra information it needs for its own, specific application. This is part of the value of inheritance; it allows the reuse of code.

This example illustrates one other important point: **super()** always refers to the constructor in the closest superclass. The **super()** in **Shipment** calls the constructor in **BoxWeight**. The **super()** in **BoxWeight** calls the constructor in **Box**. In a class hierarchy, if a superclass constructor requires parameters, then all subclasses must pass those parameters “up the line.” This is true whether or not a subclass needs parameters of its own.

NOTE In the preceding program, the entire class hierarchy, including **Box**, **BoxWeight**, and **Shipment**, is shown all in one file. This is for your convenience only. In Java, all three classes could have been placed into their own files and compiled separately. In fact, using separate files is the norm, not the exception, in creating class hierarchies.

When Constructors Are Executed

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy executed? For example, given a subclass called **B** and a superclass called **A**, is **A**'s constructor executed before **B**'s, or vice versa? The answer is that in a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass. Further, since **super()** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super()** is used. If **super()** is not used, then the default or parameterless constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```

// Demonstrate when constructors are executed.

// Create a super class.
class A {
    A() {
        System.out.println("Inside A's constructor.");
    }
}

```

```
// Create a subclass by extending class A.
class B extends A {
    B() {
        System.out.println("Inside B's constructor.");
    }
}

// Create another subclass by extending B.
class C extends B {
    C() {
        System.out.println("Inside C's constructor.");
    }
}

class CallingCons {
    public static void main(String args[]) {
        C c = new C();
    }
}
```

The output from this program is shown here:

```
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

As you can see, the constructors are executed in order of derivation.

If you think about it, it makes sense that constructors complete their execution in order of derivation. Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must complete its execution first.

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden. Consider the following:

```
// Method overriding.
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }

    // display i and j
    void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}
```

```

class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    // display k - this overrides show() in A
    void show() {
        System.out.println("k: " + k);
    }
}

class Override {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);

        subOb.show(); // this calls show() in B
    }
}

```

The output produced by this program is shown here:

```
k: 3
```

When **show()** is invoked on an object of type **B**, the version of **show()** defined within **B** is used. That is, the version of **show()** inside **B** overrides the version declared in **A**.

If you wish to access the superclass version of an overridden method, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show()** is invoked within the subclass' version. This allows all instance variables to be displayed.

```

class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    void show() {
        super.show(); // this calls A's show()
        System.out.println("k: " + k);
    }
}

```

If you substitute this version of **A** into the previous program, you will see the following output:

```
i and j: 1 2
k: 3
```

Here, **super.show()** calls the superclass version of **show()**.

Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded. For example, consider this modified version of the preceding example:

```
// Methods with differing type signatures are overloaded - not
// overridden.
class A {
    int i, j;

    A(int a, int b) {
        i = a;
        j = b;
    }

    // display i and j
    void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}

// Create a subclass by extending class A.
class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    // overload show()
    void show(String msg) {
        System.out.println(msg + k);
    }
}

class Override {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);

        subOb.show("This is k: "); // this calls show() in B
        subOb.show(); // this calls show() in A
    }
}
```

The output produced by this program is shown here:

```
This is k: 3
i and j: 1 2
```

The version of **show()** in **B** takes a string parameter. This makes its type signature different from the one in **A**, which takes no parameters. Therefore, no overriding (or name hiding) takes place. Instead, the version of **show()** in **B** simply overloads the version of **show()** in **A**.

Dynamic Method Dispatch

While the examples in the preceding section demonstrate the mechanics of method overriding, they do not show its power. Indeed, if there were nothing more to method overriding than a name space convention, then it would be, at best, an interesting curiosity, but of little real value. However, this is not the case. Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch*. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Let's begin by restating an important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Here is an example that illustrates dynamic method dispatch:

```
// Dynamic Method Dispatch
class A {
    void callme() {
        System.out.println("Inside A's callme method");
    }
}

class B extends A {
    // override callme()
    void callme() {
        System.out.println("Inside B's callme method");
    }
}

class C extends A {
    // override callme()
    void callme() {
        System.out.println("Inside C's callme method");
    }
}

class Dispatch {
    public static void main(String args[]) {
        A a = new A(); // object of type A
        B b = new B(); // object of type B
        C c = new C(); // object of type C
```

```

A r; // obtain a reference of type A

r = a; // r refers to an A object
r.callme(); // calls A's version of callme

r = b; // r refers to a B object
r.callme(); // calls B's version of callme

r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
}

```

The output from the program is shown here:

```

Inside A's callme method
Inside B's callme method
Inside C's callme method

```

This program creates one superclass called **A** and two subclasses of it, called **B** and **C**. Subclasses **B** and **C** override **callme()** declared in **A**. Inside the **main()** method, objects of type **A**, **B**, and **C** are declared. Also, a reference of type **A**, called **r**, is declared. The program then in turn assigns a reference to each type of object to **r** and uses that reference to invoke **callme()**. As the output shows, the version of **callme()** executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, **r**, you would see three calls to **A**'s **callme()** method.

NOTE Readers familiar with C++ or C# will recognize that overridden methods in Java are similar to virtual functions in those languages.

Why Overridden Methods?

As stated earlier, overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.

Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface. Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.

Dynamic, run-time polymorphism is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness. The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

Applying Method Overriding

Let's look at a more practical example that uses method overriding. The following program creates a superclass called **Figure** that stores the dimensions of a two-dimensional object. It also defines a method called **area()** that computes the area of an object. The program derives two subclasses from **Figure**. The first is **Rectangle** and the second is **Triangle**. Each of these subclasses overrides **area()** so that it returns the area of a rectangle and a triangle, respectively.

```
// Using run-time polymorphism.
class Figure {
    double dim1;
    double dim2;

    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    double area() {
        System.out.println("Area for Figure is undefined.");
        return 0;
    }
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // override area for rectangle
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}

class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // override area for right triangle
    double area() {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}

class FindAreas {
    public static void main(String args[]) {
        Figure f = new Figure(10, 10);
        Rectangle r = new Rectangle(9, 5);
```

```

Triangle t = new Triangle(10, 8);
Figure figref;

figref = r;
System.out.println("Area is " + figref.area());

figref = t;
System.out.println("Area is " + figref.area());

figref = f;
System.out.println("Area is " + figref.area());
}
}

```

The output from the program is shown here:

```

Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0

```

Through the dual mechanisms of inheritance and run-time polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects. In this case, if an object is derived from **Figure**, then its area can be obtained by calling **area()**. The interface to this operation is the same no matter what type of figure is being used.

Using Abstract Classes

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. This is the case with the class **Figure** used in the preceding example. The definition of **area()** is simply a placeholder. It will not compute and display the area of any type of object.

As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have methods that must be overridden by the subclass in order for the subclass to have any meaning. Consider the class **Triangle**. It has no meaning if **area()** is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the *abstract method*.

You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass. To declare an abstract method, use this general form:

```
abstract type name(parameter-list);
```

As you can see, no method body is present.

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared **abstract** itself.

Here is a simple example of a class with an abstract method, followed by a class which implements that method:

```
// A Simple demonstration of abstract.
abstract class A {
    abstract void callme();

    // concrete methods are still allowed in abstract classes
    void callmetoo() {
        System.out.println("This is a concrete method.");
    }
}

class B extends A {
    void callme() {
        System.out.println("B's implementation of callme.");
    }
}

class AbstractDemo {
    public static void main(String args[]) {
        B b = new B();

        b.callme();
        b.callmetoo();
    }
}
```

Notice that no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class **A** implements a concrete method called **callmetoo()**. This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object. You will see this feature put to use in the next example.

Using an abstract class, you can improve the **Figure** class shown earlier. Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the program declares **area()** as abstract inside **Figure**. This, of course, means that all classes derived from **Figure** must override **area()**.

```
// Using abstract methods and classes.
abstract class Figure {
    double dim1;
    double dim2;

    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    // area is now an abstract method
    abstract double area();
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // override area for rectangle
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}

class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // override area for right triangle
    double area() {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}

class AbstractAreas {
    public static void main(String args[]) {
        // Figure f = new Figure(10, 10); // illegal now
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);
        Figure figref; // this is OK, no object is created

        figref = r;
        System.out.println("Area is " + figref.area());

        figref = t;
```

```

        System.out.println("Area is " + figref.area());
    }
}

```

As the comment inside `main()` indicates, it is no longer possible to declare objects of type `Figure`, since it is now abstract. And, all subclasses of `Figure` must override `area()`. To prove this to yourself, try creating a subclass that does not override `area()`. You will receive a compile-time error.

Although it is not possible to create an object of type `Figure`, you can create a reference variable of type `Figure`. The variable `figref` is declared as a reference to `Figure`, which means that it can be used to refer to an object of any class derived from `Figure`. As explained, it is through superclass reference variables that overridden methods are resolved at run time.

Using final with Inheritance

The keyword `final` has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of `final` apply to inheritance. Both are examined here.

Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify `final` as a modifier at the start of its declaration. Methods declared as `final` cannot be overridden. The following fragment illustrates `final`:

```

class A {
    final void meth() {
        System.out.println("This is a final method.");
    }
}

class B extends A {
    void meth() { // ERROR! Can't override.
        System.out.println("Illegal!");
    }
}

```

Because `meth()` is declared as `final`, it cannot be overridden in `B`. If you attempt to do so, a compile-time error will result.

Methods declared as `final` can sometimes provide a performance enhancement: The compiler is free to *inline* calls to them because it “knows” they will not be overridden by a subclass. When a small `final` method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is an option only with `final` methods. Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*. However, since `final` methods cannot be overridden, a call to one can be resolved at compile time. This is called *early binding*.

Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a **final** class:

```
final class A {
    //...
}

// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
    //...
}
```

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

The Object Class

There is one special class, **Object**, defined by Java. All other classes are subclasses of **Object**. That is, **Object** is a superclass of all other classes. This means that a reference variable of type **Object** can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type **Object** can also refer to any array.

Object defines the following methods, which means that they are available in every object.

Method	Purpose
<code>Object clone()</code>	Creates a new object that is the same as the object being cloned.
<code>boolean equals(Object object)</code>	Determines whether one object is equal to another.
<code>void finalize()</code>	Called before an unused object is recycled.
<code>Class<?> getClass()</code>	Obtains the class of an object at run time.
<code>int hashCode()</code>	Returns the hash code associated with the invoking object.
<code>void notify()</code>	Resumes execution of a thread waiting on the invoking object.
<code>void notifyAll()</code>	Resumes execution of all threads waiting on the invoking object.
<code>String toString()</code>	Returns a string that describes the object.
<code>void wait()</code> <code>void wait(long milliseconds)</code> <code>void wait(long milliseconds, int nanoseconds)</code>	Waits on another thread of execution.

The methods `getClass()`, `notify()`, `notifyAll()`, and `wait()` are declared as `final`. You may override the others. These methods are described elsewhere in this book. However, notice two methods now: `equals()` and `toString()`. The `equals()` method compares two objects. It returns `true` if the objects are equal, and `false` otherwise. The precise definition of equality can vary, depending on the type of objects being compared. The `toString()` method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using `println()`. Many classes override this method. Doing so allows them to tailor a description specifically for the types of objects that they create.

One last point: Notice the unusual syntax in the return type for `getClass()`. This relates to Java's *generics* feature, which is described in Chapter 14.

CHAPTER

9

Packages and Interfaces

This chapter examines two of Java’s most innovative features: packages and interfaces. *Packages* are containers for classes. They are used to keep the class name space compartmentalized. For example, a package allows you to create a class named **List**, which you can store in your own package without concern that it will collide with some other class named **List** stored elsewhere. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

In previous chapters, you have seen how methods define the interface to the data in a class. Through the use of the **interface** keyword, Java allows you to fully abstract an interface from its implementation. Using **interface**, you can specify a set of methods that can be implemented by one or more classes. In its traditional form, the **interface**, itself, does not actually define any implementation. Although they are similar to abstract classes, **interfaces** have an additional capability: A class can implement more than one interface. By contrast, a class can only inherit a single superclass (abstract or otherwise).

Packages

In the preceding chapters, the name of each example class was taken from the same name space. This means that a unique name had to be used for each class to avoid name collisions. After a while, without some way to manage the name space, you could run out of convenient, descriptive names for individual classes. You also need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers. (Imagine a small group of programmers fighting over who gets to use the name “Foobar” as a class name. Or, imagine the entire Internet community arguing over who first named a class “Espresso.”) Thankfully, Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are exposed only to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

Defining a Package

To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name. (This is why you haven't had to worry about packages before now.) While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

This is the general form of the **package** statement:

```
package pkg;
```

Here, *pkg* is the name of the package. For example, the following statement creates a package called **MyPackage**:

```
package MyPackage;
```

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. Remember that case is significant, and the directory name must match the package name exactly.

More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

```
package pkg1[.pkg2[.pkg3]];
```

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

```
package java.awt.image;
```

needs to be stored in **java\awt\image** in a Windows environment. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH

As just explained, packages are mirrored by directories. This raises an important question: How does the Java run-time system know where to look for packages that you create? The answer has three parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable. Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.

For example, consider the following package specification:

```
package MyPack
```

In order for a program to find **MyPack**, one of three things must be true. Either the program can be executed from a directory immediately above **MyPack**, or the **CLASSPATH** must be set to include the path to **MyPack**, or the **-classpath** option must specify the path to **MyPack** when the program is run via **java**.

When the second two options are used, the class path *must not* include **MyPack**, itself. It must simply specify the *path to MyPack*. For example, in a Windows environment, if the path to **MyPack** is

```
C:\MyPrograms\Java\MyPack
```

then the class path to **MyPack** is

```
C:\MyPrograms\Java
```

The easiest way to try the examples shown in this book is to simply create the package directories below your current development directory, put the **.class** files into the appropriate directories, and then execute the programs from the development directory. This is the approach used in the following example.

A Short Package Example

Keeping the preceding discussion in mind, you can try this simple package:

```
// A simple package
package MyPack;

class Balance {
    String name;
    double bal;

    Balance(String n, double b) {
        name = n;
        bal = b;
    }

    void show() {
        if(bal<0)
            System.out.print("-> ");
        System.out.println(name + ": $" + bal);
    }
}

class AccountBalance {
    public static void main(String args[]) {
        Balance current[] = new Balance[3];

        current[0] = new Balance("K. J. Fielding", 123.23);
        current[1] = new Balance("Will Tell", 157.02);
        current[2] = new Balance("Tom Jackson", -12.33);
```

```

        for(int i=0; i<3; i++) current[i].show();
    }
}

```

Call this file **AccountBalance.java** and put it in a directory called **MyPack**.

Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then, try executing the **AccountBalance** class, using the following command line:

```
java MyPack.AccountBalance
```

Remember, you will need to be in the directory above **MyPack** when you execute this command. (Alternatively, you can use one of the other two options described in the preceding section to specify the path **MyPack**.)

As explained, **AccountBalance** is now part of the package **MyPack**. This means that it cannot be executed by itself. That is, you cannot use this command line:

```
java AccountBalance
```

AccountBalance must be qualified with its package name.

Access Protection

In the preceding chapters, you learned about various aspects of Java's access control mechanism and its access modifiers. For example, you already know that access to a **private** member of a class is granted only to other members of that class. Packages add another dimension to access control. As you will see, Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three access modifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories. Table 9-1 sums up the interactions.

While Java's access control mechanism may seem complicated, we can simplify it as follows. Anything declared **public** can be accessed from anywhere. Anything declared **private** cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Table 9-1 Class Member Access

Table 9-1 applies only to members of classes. A non-nested class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.

An Access Example

The following example shows all combinations of the access control modifiers. This example has two packages and five classes. Remember that the classes for the two different packages need to be stored in directories named after their respective packages—in this case, **p1** and **p2**.

The source for the first package defines three classes: **Protection**, **Derived**, and **SamePackage**. The first class defines four **int** variables in each of the legal protection modes. The variable **n** is declared with the default protection, **n_pri** is **private**, **n_pro** is **protected**, and **n_pub** is **public**.

Each subsequent class in this example will try to access the variables in an instance of this class. The lines that will not compile due to access restrictions are commented out. Before each of these lines is a comment listing the places from which this level of protection would allow access.

The second class, **Derived**, is a subclass of **Protection** in the same package, **p1**. This grants **Derived** access to every variable in **Protection** except for **n_pri**, the **private** one. The third class, **SamePackage**, is not a subclass of **Protection**, but is in the same package and also has access to all but **n_pri**.

This is file **Protection.java**:

```
package p1;

public class Protection {
    int n = 1;
    private int n_pri = 2;
    protected int n_pro = 3;
    public int n_pub = 4;

    public Protection() {
        System.out.println("base constructor");
        System.out.println("n = " + n);
        System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
```

```

        System.out.println("n_pub = " + n_pub);
    }
}

```

This is file **Derived.java**:

```

package p1;

class Derived extends Protection {
    Derived() {
        System.out.println("derived constructor");
        System.out.println("n = " + n);

        // class only
        // System.out.println("n_pri = " + n_pri);

        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}

```

This is file **SamePackage.java**:

```

package p1;

class SamePackage {
    SamePackage() {
        Protection p = new Protection();
        System.out.println("same package constructor");
        System.out.println("n = " + p.n);

        // class only
        // System.out.println("n_pri = " + p.n_pri);

        System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}

```

Following is the source code for the other package, **p2**. The two classes defined in **p2** cover the other two conditions that are affected by access control. The first class, **Protection2**, is a subclass of **p1.Protection**. This grants access to all of **p1.Protection**'s variables except for **n_pri** (because it is **private**) and **n**, the variable declared with the default protection. Remember, the default only allows access from within the class or the package, not extra-package subclasses. Finally, the class **OtherPackage** has access to only one variable, **n_pub**, which was declared **public**.

This is file **Protection2.java**:

```

package p2;

class Protection2 extends p1.Protection {
    Protection2() {

```

```

        System.out.println("derived other package constructor");

// class or package only
// System.out.println("n = " + n);

// class only
// System.out.println("n_pri = " + n_pri);

        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}

```

This is file **OtherPackage.java**:

```

package p2;

class OtherPackage {
    OtherPackage() {
        p1.Protection p = new p1.Protection();
        System.out.println("other package constructor");

// class or package only
// System.out.println("n = " + p.n);

// class only
// System.out.println("n_pri = " + p.n_pri);

// class, subclass or package only
// System.out.println("n_pro = " + p.n_pro);

        System.out.println("n_pub = " + p.n_pub);
    }
}

```

If you want to try these two packages, here are two test files you can use. The one for package **p1** is shown here:

```

// Demo package p1.
package p1;

// Instantiate the various classes in p1.
public class Demo {
    public static void main(String args[]) {
        Protection ob1 = new Protection();
        Derived ob2 = new Derived();
        SamePackage ob3 = new SamePackage();
    }
}

```

The test file for **p2** is shown next:

```
// Demo package p2.
package p2;
```

```
// Instantiate the various classes in p2.
public class Demo {
    public static void main(String args[]) {
        Protection2 ob1 = new Protection2();
        OtherPackage ob2 = new OtherPackage();
    }
}
```

Importing Packages

Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages. There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use. For this reason, Java includes the **import** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.

In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. This is the general form of the **import** statement:

```
import pkg1 [.pkg2].(classname | *);
```

Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit *classname* or a star (*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use:

```
import java.util.Date;
import java.io.*;
```

All of the standard Java classes included with Java are stored in a package called **java**. The basic language functions are stored in a package inside of the **java** package called **java.lang**. Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in **java.lang**, it is implicitly imported by the compiler for all programs. This is equivalent to the following line being at the top of all of your programs:

```
import java.lang.*;
```

If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile-time error and have to explicitly name the class specifying its package.

It must be emphasized that the **import** statement is optional. Any place you use a class name, you can use its *fully qualified name*, which includes its full package hierarchy. For example, this fragment uses an import statement:

```
import java.util.*;
class MyDate extends Date {
```

The same example without the **import** statement looks like this:

```
class MyDate extends java.util.Date {
```

In this version, **Date** is fully-qualified.

As shown in Table 9-1, when a package is imported, only those items within the package declared as **public** will be available to non-subclasses in the importing code. For example, if you want the **Balance** class of the package **MyPack** shown earlier to be available as a stand-alone class for general use outside of **MyPack**, then you will need to declare it as **public** and put it into its own file, as shown here:

```
package MyPack;

/* Now, the Balance class, its constructor, and its
   show() method are public. This means that they can
   be used by non-subclass code outside their package.
*/
public class Balance {
    String name;
    double bal;

    public Balance(String n, double b) {
        name = n;
        bal = b;
    }

    public void show() {
        if(bal<0)
            System.out.print("-> ");
        System.out.println(name + ": $" + bal);
    }
}
```

As you can see, the **Balance** class is now **public**. Also, its constructor and its **show()** method are **public**, too. This means that they can be accessed by any type of code outside the **MyPack** package. For example, here **TestBalance** imports **MyPack** and is then able to make use of the **Balance** class:

```
import MyPack.*;

class TestBalance {
    public static void main(String args[]) {
```

```

    /* Because Balance is public, you may use Balance
       class and call its constructor. */
    Balance test = new Balance("J. J. Jaspers", 99.88);

    test.show(); // you may also call show()
}
}

```

As an experiment, remove the **public** specifier from the **Balance** class and then try compiling **TestBalance**. As explained, errors will result.

Interfaces

Using the keyword **interface**, you can fully abstract a class' interface from its implementation. That is, using **interface**, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body. In practice, this means that you can define interfaces that don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an **interface**. Also, one class can implement any number of interfaces.

To implement an interface, a class must provide the complete set of methods required by the interface. However, each class is free to determine the details of its own implementation. By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and nonextensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

Defining an Interface

An interface is defined much like a class. This is a simplified general form of an interface:

```

access interface name {
    return-type method-name1(parameter-list);
    return-type method-name2(parameter-list);

    type final-varname1 = value;
    type final-varname2 = value;
    // ...
    return-type method-nameN(parameter-list);
    type final-varnameN = value;
}

```

When no access modifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface. *name* is the name of the interface, and can be any valid identifier. Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods. Each class that includes such an interface must implement all of the methods.

Before continuing an important point needs to be made. JDK 8 added a feature to **interface** that makes a significant change to its capabilities. Prior to JDK 8, an interface could not define any implementation whatsoever. This is the type of interface that the preceding simplified form shows, in which no method declaration supplies a body. Thus, prior to JDK 8, an interface could define only “what,” but not “how.” JDK 8 changes this. Beginning with JDK 8, it is possible to add a *default implementation* to an interface method. Thus, it is now possible for **interface** to specify some behavior. However, default methods constitute what is, in essence, a special-use feature, and the original intent behind **interface** still remains. Therefore, as a general rule, you will still often create and use interfaces in which no default methods exist. For this reason, we will begin by discussing the interface in its traditional form. The default method is described at the end of this chapter.

As the general form shows, variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly **public**.

Here is an example of an interface definition. It declares a simple interface that contains one method called **callback()** that takes a single integer parameter.

```
interface Callback {
    void callback(int param);
}
```

Implementing Interfaces

Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface, include the **implements** clause in a class definition, and then create the methods required by the interface. The general form of a class that includes the **implements** clause looks like this:

```
class classname [extends superclass] [implements interface [,interface...]] {
    // class-body
}
```

If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared **public**. Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

Here is a small example class that implements the **Callback** interface shown earlier:

```
class Client implements Callback {
    // Implement Callback's interface
```

```

public void callback(int p) {
    System.out.println("callback called with " + p);
}
}

```

Notice that **callback()** is declared using the **public** access modifier.

REMEMBER When you implement an interface method, it must be declared as **public**.

It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of **Client** implements **callback()** and adds the method **nonIfaceMeth()**:

```

class Client implements Callback {
    // Implement Callback's interface
    public void callback(int p) {
        System.out.println("callback called with " + p);
    }

    void nonIfaceMeth() {
        System.out.println("Classes that implement interfaces " +
                           "may also define other members, too.");
    }
}

```

Accessing Implementations Through Interface References

You can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the “callee.” This process is similar to using a superclass reference to access a subclass object, as described in Chapter 8.

CAUTION Because dynamic lookup of a method at run time incurs a significant overhead when compared with the normal method invocation in Java, you should be careful not to use interfaces casually in performance-critical code.

The following example calls the **callback()** method via an interface reference variable:

```

class TestIface {
    public static void main(String args[]) {
        Callback c = new Client();
        c.callback(42);
    }
}

```

The output of this program is shown here:

```
callback called with 42
```

Notice that variable **c** is declared to be of the interface type **Callback**, yet it was assigned an instance of **Client**. Although **c** can be used to access the **callback()** method, it cannot access any other members of the **Client** class. An interface reference variable has knowledge only of the methods declared by its **interface** declaration. Thus, **c** could not be used to access **nonIFaceMeth()** since it is defined by **Client** but not **Callback**.

While the preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference. To sample this usage, first create the second implementation of **Callback**, shown here:

```
// Another implementation of Callback.
class AnotherClient implements Callback {
    // Implement Callback's interface
    public void callback(int p) {
        System.out.println("Another version of callback");
        System.out.println("p squared is " + (p*p));
    }
}
```

Now, try the following class:

```
class TestIface2 {
    public static void main(String args[]) {
        Callback c = new Client();
        AnotherClient ob = new AnotherClient();

        c.callback(42);

        c = ob; // c now refers to AnotherClient object
        c.callback(42);
    }
}
```

The output from this program is shown here:

```
callback called with 42
Another version of callback
p squared is 1764
```

As you can see, the version of **callback()** that is called is determined by the type of object that **c** refers to at run time. While this is a very simple example, you will see another, more practical one shortly.

Partial Implementations

If a class includes an interface but does not fully implement the methods required by that interface, then that class must be declared as **abstract**. For example:

```
abstract class Incomplete implements Callback {
    int a, b;
```

```

void show() {
    System.out.println(a + " " + b);
}
//...
}

```

Here, the class **Incomplete** does not implement **callback()** and must be declared as **abstract**. Any class that inherits **Incomplete** must implement **callback()** or be declared **abstract** itself.

Nested Interfaces

An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*. A nested interface can be declared as **public**, **private**, or **protected**. This differs from a top-level interface, which must either be declared as **public** or use the default access level, as previously described. When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member. Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

Here is an example that demonstrates a nested interface:

```

// A nested interface example.

// This class contains a member interface.
class A {
    // this is a nested interface
    public interface NestedIF {
        boolean isNotNegative(int x);
    }
}

// B implements the nested interface.
class B implements A.NestedIF {
    public boolean isNotNegative(int x) {
        return x < 0 ? false: true;
    }
}

class NestedIFDemo {
    public static void main(String args[]) {
        // use a nested interface reference
        A.NestedIF nif = new B();

        if(nif.isNotNegative(10))
            System.out.println("10 is not negative");
        if(nif.isNotNegative(-12))
            System.out.println("this won't be displayed");
    }
}

```

Notice that **A** defines a member interface called **NestedIF** and that it is declared **public**. Next, **B** implements the nested interface by specifying

```
implements A.NestedIF
```

Notice that the name is fully qualified by the enclosing class' name. Inside the **main()** method, an **A.NestedIF** reference called **nif** is created, and it is assigned a reference to a **B** object. Because **B** implements **A.NestedIF**, this is legal.

Applying Interfaces

To understand the power of interfaces, let's look at a more practical example. In earlier chapters, you developed a class called **Stack** that implemented a simple fixed-size stack. However, there are many ways to implement a stack. For example, the stack can be of a fixed size or it can be "growable." The stack can also be held in an array, a linked list, a binary tree, and so on. No matter how the stack is implemented, the interface to the stack remains the same. That is, the methods **push()** and **pop()** define the interface to the stack independently of the details of the implementation. Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics. Let's look at two examples.

First, here is the interface that defines an integer stack. Put this in a file called **IntStack.java**. This interface will be used by both stack implementations.

```
// Define an integer stack interface.
interface IntStack {
    void push(int item); // store an item
    int pop(); // retrieve an item
}
```

The following program creates a class called **FixedStack** that implements a fixed-length version of an integer stack:

```
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
    private int stck[];
    private int tos;

    // allocate and initialize stack
    FixedStack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // Push an item onto the stack
    public void push(int item) {
        if(tos==stck.length-1) // use length member
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }
}
```

```

// Pop an item from the stack
public int pop() {
    if(tos < 0) {
        System.out.println("Stack underflow.");
        return 0;
    }
    else
        return stck[tos--];
}
}

class IFTest {
    public static void main(String args[]) {
        FixedStack mystack1 = new FixedStack(5);
        FixedStack mystack2 = new FixedStack(8);

        // push some numbers onto the stack
        for(int i=0; i<5; i++) mystack1.push(i);
        for(int i=0; i<8; i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());

        System.out.println("Stack in mystack2:");
        for(int i=0; i<8; i++)
            System.out.println(mystack2.pop());
    }
}

```

Following is another implementation of **IntStack** that creates a dynamic stack by use of the same **interface** definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled.

```

// Implement a "growable" stack.
class DynStack implements IntStack {
    private int stck[];
    private int tos;

    // allocate and initialize stack
    DynStack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // Push an item onto the stack
    public void push(int item) {
        // if stack is full, allocate a larger stack
        if(tos==stck.length-1) {
            int temp[] = new int[stck.length * 2]; // double size
            for(int i=0; i<stck.length; i++) temp[i] = stck[i];
            stck = temp;
            tos++;
        }
        stck[tos] = item;
        tos++;
    }
}

```

```

        stck = temp;
        stck[++tos] = item;
    }
    else
        stck[++tos] = item;
}

// Pop an item from the stack
public int pop() {
    if(tos < 0) {
        System.out.println("Stack underflow.");
        return 0;
    }
    else
        return stck[tos--];
}

class IFTest2 {
    public static void main(String args[]) {
        DynStack mystack1 = new DynStack(5);
        DynStack mystack2 = new DynStack(8);

        // these loops cause each stack to grow
        for(int i=0; i<12; i++) mystack1.push(i);
        for(int i=0; i<20; i++) mystack2.push(i);

        System.out.println("Stack in mystack1:");
        for(int i=0; i<12; i++)
            System.out.println(mystack1.pop());

        System.out.println("Stack in mystack2:");
        for(int i=0; i<20; i++)
            System.out.println(mystack2.pop());
    }
}

```

The following class uses both the **FixedStack** and **DynStack** implementations. It does so through an interface reference. This means that calls to **push()** and **pop()** are resolved at run time rather than at compile time.

```

/*
 * Create an interface variable and
 * access stacks through it.
 */
class IFTest3 {
    public static void main(String args[]) {
        IntStack mystack; // create an interface reference variable
        DynStack ds = new DynStack(5);
        FixedStack fs = new FixedStack(8);

        mystack = ds; // load dynamic stack
        // push some numbers onto the stack
        for(int i=0; i<12; i++) mystack.push(i);

```

```

mystack = fs; // load fixed stack
for(int i=0; i<8; i++) mystack.push(i);

mystack = ds;
System.out.println("Values in dynamic stack:");
for(int i=0; i<12; i++)
    System.out.println(mystack.pop());

mystack = fs;
System.out.println("Values in fixed stack:");
for(int i=0; i<8; i++)
    System.out.println(mystack.pop());
}
}

```

In this program, **mystack** is a reference to the **IntStack** interface. Thus, when it refers to **ds**, it uses the versions of **push()** and **pop()** defined by the **DynStack** implementation. When it refers to **fs**, it uses the versions of **push()** and **pop()** defined by **FixedStack**. As explained, these determinations are made at run time. Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

Variables in Interfaces

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When you include that interface in a class (that is, when you “implement” the interface), all of those variable names will be in scope as constants. (This is similar to using a header file in C/C++ to create a large number of **#defined** constants or **const** declarations.) If an interface contains no methods, then any class that includes such an interface doesn’t actually implement anything. It is as if that class were importing the constant fields into the class name space as **final** variables. The next example uses this technique to implement an automated “decision maker”:

```

import java.util.Random;

interface SharedConstants {
    int NO = 0;
    int YES = 1;
    int MAYBE = 2;
    int LATER = 3;
    int SOON = 4;
    int NEVER = 5;
}

class Question implements SharedConstants {
    Random rand = new Random();
    int ask() {
        int prob = (int) (100 * rand.nextDouble());
        if (prob < 30)

```

```

        return NO;           // 30%
    else if (prob < 60)
        return YES;         // 30%
    else if (prob < 75)
        return LATER;       // 15%
    else if (prob < 98)
        return SOON;        // 13%

    else
        return NEVER;       // 2%
}
}

class AskMe implements SharedConstants {
    static void answer(int result) {
        switch(result) {
            case NO:
                System.out.println("No");
                break;
            case YES:
                System.out.println("Yes");
                break;
            case MAYBE:
                System.out.println("Maybe");
                break;
            case LATER:
                System.out.println("Later");
                break;
            case SOON:
                System.out.println("Soon");
                break;
            case NEVER:
                System.out.println("Never");
                break;
        }
    }

    public static void main(String args[]) {
        Question q = new Question();

        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
    }
}

```

Notice that this program makes use of one of Java's standard classes: **Random**. This class provides pseudorandom numbers. It contains several methods that allow you to obtain random numbers in the form required by your program. In this example, the method **nextDouble()** is used. It returns random numbers in the range 0.0 to 1.0.

In this sample program, the two classes, **Question** and **AskMe**, both implement the **SharedConstants** interface where **NO**, **YES**, **MAYBE**, **SOON**, **LATER**, and **NEVER** are

defined. Inside each class, the code refers to these constants as if each class had defined or inherited them directly. Here is the output of a sample run of this program. Note that the results are different each time it is run.

```
Later
Soon
No
Yes
```

NOTE The technique of using an interface to define shared constants, as just described, is controversial. It is described here for completeness.

Interfaces Can Be Extended

One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain. Following is an example:

```
// One interface can extend another.
interface A {
    void meth1();
    void meth2();
}

// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
    void meth3();
}

// This class must implement all of A and B
class MyClass implements B {
    public void meth1() {
        System.out.println("Implement meth1().");
    }

    public void meth2() {
        System.out.println("Implement meth2().");
    }

    public void meth3() {
        System.out.println("Implement meth3().");
    }
}

class IFExtend {
    public static void main(String arg[]) {
        MyClass ob = new MyClass();
```

```
    ob.meth1();  
    ob.meth2();  
    ob.meth3();  
}  
}
```

As an experiment, you might want to try removing the implementation for **meth1()** in **MyClass**. This will cause a compile-time error. As stated earlier, any class that implements an interface must implement all methods required by that interface, including any that are inherited from other interfaces.

Default Interface Methods

As explained earlier, prior to JDK 8, an interface could not define any implementation whatsoever. This meant that for all previous versions of Java, the methods specified by an interface were abstract, containing no body. This is the traditional form of an interface and is the type of interface that the preceding discussions have used. The release of JDK 8 has changed this by adding a new capability to **interface** called the *default method*. A default method lets you define a default implementation for an interface method. In other words, by use of a default method, it is now possible for an interface method to provide a body, rather than being abstract. During its development, the default method was also referred to as an *extension method*, and you will likely see both terms used.

A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code. Recall that there must be implementations for all methods defined by an interface. In the past, if a new method were added to a popular, widely used interface, then the addition of that method would break existing code because no implementation would be found for that new method. The default method solves this problem by supplying an implementation that will be used if no other implementation is explicitly provided. Thus, the addition of a default method will not cause preexisting code to break.

Another motivation for the default method was the desire to specify methods in an interface that are, essentially, optional, depending on how the interface is used. For example, an interface might define a group of methods that act on a sequence of elements. One of these methods might be called **remove()**, and its purpose is to remove an element from the sequence. However, if the interface is intended to support both modifiable and nonmodifiable sequences, then **remove()** is essentially optional because it won't be used by nonmodifiable sequences. In the past, a class that implemented a nonmodifiable sequence would have had to define an empty implementation of **remove()**, even though it was not needed. Today, a default implementation for **remove()** can be specified in the interface that does nothing (or throws an exception). Providing this default prevents a class used for nonmodifiable sequences from having to define its own, placeholder version of **remove()**. Thus, by providing a default, the interface makes the implementation of **remove()** by a class optional.

It is important to point out that the addition of default methods does not change a key aspect of **interface**: its inability to maintain state information. An interface still cannot have instance variables, for example. Thus, the defining difference between an interface and a class is that a class can maintain state information, but an interface cannot. Furthermore, it

is still not possible to create an instance of an interface by itself. It must be implemented by a class. Therefore, even though, beginning with JDK 8, an interface can define default methods, the interface must still be implemented by a class if an instance is to be created.

One last point: As a general rule, default methods constitute a special-purpose feature. Interfaces that you create will still be used primarily to specify *what* and not *how*. However, the inclusion of the default method gives you added flexibility.

Default Method Fundamentals

An interface default method is defined similar to the way a method is defined by a **class**. The primary difference is that the declaration is preceded by the keyword **default**. For example, consider this simple interface:

```
public interface MyIF {
    // This is a "normal" interface method declaration.
    // It does NOT define a default implementation.
    int getNumber();

    // This is a default method. Notice that it provides
    // a default implementation.
    default String getString() {
        return "Default String";
    }
}
```

MyIF declares two methods. The first, **getNumber()**, is a standard interface method declaration. It defines no implementation whatsoever. The second method is **getString()**, and it does include a default implementation. In this case, it simply returns the string "Default String". Pay special attention to the way **getString()** is declared. Its declaration is preceded by the **default** modifier. This syntax can be generalized. To define a default method, precede its declaration with **default**.

Because **getString()** includes a default implementation, it is not necessary for an implementing class to override it. In other words, if an implementing class does not provide its own implementation, the default is used. For example, the **MyIFImp** class shown next is perfectly valid:

```
// Implement MyIF.
class MyIFImp implements MyIF {
    // Only getNumber() defined by MyIF needs to be implemented.
    // getString() can be allowed to default.
    public int getNumber() {
        return 100;
    }
}
```

The following code creates an instance of **MyIFImp** and uses it to call both **getNumber()** and **getString()**.

```
// Use the default method.
class DefaultMethodDemo {
```

```

public static void main(String args[]) {
    MyIFImp obj = new MyIFImp();

    // Can call getNumber(), because it is explicitly
    // implemented by MyIFImp:
    System.out.println(obj.getNumber());

    // Can also call getString(), because of default
    // implementation:
    System.out.println(obj.getString());
}
}

```

The output is shown here:

```

100
Default String

```

As you can see, the default implementation of `getString()` was automatically used. It was not necessary for `MyIFImp` to define it. Thus, for `getString()`, implementation by a class is optional. (Of course, its implementation by a class will be *required* if the class uses `getString()` for some purpose beyond that supported by its default.)

It is both possible and common for an implementing class to define its own implementation of a default method. For example, `MyIFImp2` overrides `getString()`:

```

class MyIFImp2 implements MyIF {
    // Here, implementations for both getNumber() and getString() are provided.
    public int getNumber() {
        return 100;
    }

    public String getString() {
        return "This is a different string.";
    }
}

```

Now, when `getString()` is called, a different string is returned.

A More Practical Example

Although the preceding shows the mechanics of using default methods, it doesn't illustrate their usefulness in a more practical setting. To do this, let's once again return to the `IntStack` interface shown earlier in this chapter. For the sake of discussion, assume that `IntStack` is widely used and many programs rely on it. Further assume that we now want to add a method to `IntStack` that clears the stack, enabling the stack to be re-used. Thus, we want to evolve the `IntStack` interface so that it defines new functionality, but we don't want to break any preexisting code. In the past, this would be impossible, but with the inclusion

of default methods, it is now easy to do. For example, the **IntStack** interface can be enhanced like this:

```
interface IntStack {  
    void push(int item); // store an item  
    int pop(); // retrieve an item  
  
    // Because clear() has a default, it need not be  
    // implemented by a preexisting class that uses IntStack.  
    default void clear() {  
        System.out.println("clear() not implemented.");  
    }  
}
```

Here, the default behavior of **clear()** simply displays a message indicating that it is not implemented. This is acceptable because no preexisting class that implements **IntStack** would ever call **clear()** because it was not defined by the earlier version of **IntStack**. However, **clear()** can be implemented by a new class that implements **IntStack**. Furthermore, **clear()** needs to be defined by a new implementation only if it is used. Thus, the default method gives you

- a way to gracefully evolve interfaces over time, and
- a way to provide optional functionality without requiring that a class provide a placeholder implementation when that functionality is not needed.

One other point: In real-world code, **clear()** would have thrown an exception, rather than displaying an error message. Exceptions are described in the next chapter. After working through that material, you might want to try modifying **clear()** so that its default implementation throws an **UnsupportedOperationException**.

Multiple Inheritance Issues

As explained earlier in this book, Java does not support the multiple inheritance of classes. Now that an interface can include default methods, you might be wondering if an interface can provide a way around this restriction. The answer is, essentially, no. Recall that there is still a key difference between a class and an interface: a class can maintain state information (especially through the use of instance variables), but an interface cannot.

The preceding notwithstanding, default methods do offer a bit of what one would normally associate with the concept of multiple inheritance. For example, you might have a class that implements two interfaces. If each of these interfaces provides default methods, then some behavior is inherited from both. Thus, to a limited extent, default methods do support multiple inheritance of behavior. As you might guess, in such a situation, it is possible that a name conflict will occur.

For example, assume that two interfaces called **Alpha** and **Beta** are implemented by a class called **MyClass**. What happens if both **Alpha** and **Beta** provide a method called **reset()** for which both declare a default implementation? Is the version by **Alpha** or the version by **Beta** used by **MyClass**? Or, consider a situation in which **Beta** extends **Alpha**. Which version of the default method is used? Or, what if **MyClass** provides its own implementation of the

method? To handle these and other similar types of situations, Java defines a set of rules that resolves such conflicts.

First, in all cases, a class implementation takes priority over an interface default implementation. Thus, if **MyClass** provides an override of the **reset()** default method, **MyClass'** version is used. This is the case even if **MyClass** implements both **Alpha** and **Beta**. In this case, both defaults are overridden by **MyClass'** implementation.

Second, in cases in which a class implements two interfaces that both have the same default method, but the class does not override that method, then an error will result. Continuing with the example, if **MyClass** implements both **Alpha** and **Beta**, but does not override **reset()**, then an error will occur.

In cases in which one interface inherits another, with both defining a common default method, the inheriting interface's version of the method takes precedence. Therefore, continuing the example, if **Beta** extends **Alpha**, then **Beta**'s version of **reset()** will be used.

It is possible to explicitly refer to a default implementation in an inherited interface by using a new form of **super**. Its general form is shown here:

InterfaceName.super.methodName()

For example, if **Beta** wants to refer to **Alpha**'s default for **reset()**, it can use this statement:

```
Alpha.super.reset();
```

Use static Methods in an Interface

JDK 8 added another new capability to **interface**: the ability to define one or more **static** methods. Like **static** methods in a class, a **static** method defined by an interface can be called independently of any object. Thus, no implementation of the interface is necessary, and no instance of the interface is required, in order to call a **static** method. Instead, a **static** method is called by specifying the interface name, followed by a period, followed by the method name. Here is the general form:

InterfaceName.staticMethodName

Notice that this is similar to the way that a **static** method in a class is called.

The following shows an example of a **static** method in an interface by adding one to **MyIF**, shown in the previous section. The **static** method is **getDefaultNumber()**. It returns zero.

```
public interface MyIF {
    // This is a "normal" interface method declaration.
    // It does NOT define a default implementation.
    int getNumber();

    // This is a default method. Notice that it provides
    // a default implementation.
    default String getString() {
        return "Default String";
    }
}
```

```
// This is a static interface method.  
static int getDefaultNumber() {  
    return 0;  
}  
}
```

The `getDefaultNumber()` method can be called, as shown here:

```
int defNum = MyIF.getDefaultNumber();
```

As mentioned, no implementation or instance of `MyIF` is required to call `getDefaultNumber()` because it is **static**.

One last point: **static** interface methods are not inherited by either an implementing class or a subinterface.

Final Thoughts on Packages and Interfaces

Although the examples we've included in this book do not make frequent use of packages or interfaces, both of these tools are an important part of the Java programming environment. Virtually all real programs that you write in Java will be contained within packages. A number will probably implement interfaces as well. It is important, therefore, that you be comfortable with their usage.

CHAPTER

10

Exception Handling

This chapter examines Java's exception-handling mechanism. An *exception* is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on. This approach is as cumbersome as it is troublesome. Java's exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world.

Exception-Handling Fundamentals

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

This is the general form of an exception-handling block:

```
try {  
    // block of code to monitor for errors  
}  
  
catch (ExceptionType1 exOb) {  
    // exception handler for ExceptionType1  
}  
  
catch (ExceptionType2 exOb) {  
    // exception handler for ExceptionType2  
}  
// ...  
finally {  
    // block of code to be executed after try block ends  
}
```

Here, *ExceptionType* is the type of exception that has occurred. The remainder of this chapter describes how to apply this framework.

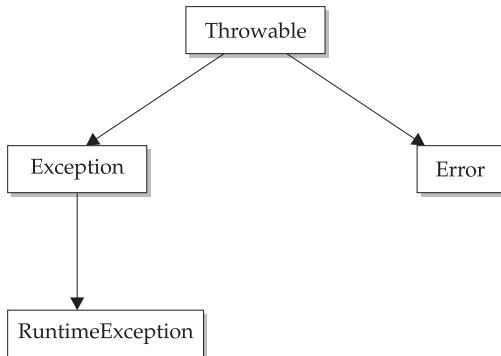
NOTE Beginning with JDK 7, there is another form of the **try** statement that supports *automatic resource management*. This form of **try**, called **try-with-resources**, is described in Chapter 13 in the context of managing files because files are some of the most commonly used resources.

Exception Types

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy. Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error. This chapter will not be dealing with exceptions of type **Error**, because these are typically created in response to catastrophic failures that cannot usually be handled by your program.

The top-level exception hierarchy is shown here:



Uncaught Exceptions

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them. This small program includes an expression that intentionally causes a divide-by-zero error:

```

class Exc0 {
    public static void main(String args[]) {
        int d = 0;
        int a = 42 / d;
    }
}
  
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the exception generated when this example is executed:

```

java.lang.ArithmaticException: / by zero
at Exc0.main(Exc0.java:4)
  
```

Notice how the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, **4**, are all included in the simple stack trace. Also, notice that the type of exception thrown is a subclass of **Exception** called **ArithmaticException**, which more specifically describes what type of error happened. As discussed later in this chapter, Java supplies several built-in exception types that match the various sorts of run-time errors that can be generated.

The stack trace will always show the sequence of method invocations that led up to the error. For example, here is another version of the preceding program that introduces the same error but in a method separate from `main()`:

```
class Exc1 {
    static void subroutine() {
        int d = 0;
        int a = 10 / d;
    }
    public static void main(String args[]) {
        Exc1.subroutine();
    }
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmaticException: / by zero
    at Exc1.subroutine(Exc1.java:4)
    at Exc1.main(Exc1.java:7)
```

As you can see, the bottom of the stack is `main`'s line 7, which is the call to `subroutine()`, which caused the exception at line 4. The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the error.

Using try and catch

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating. Most users would be confused (to say the least) if your program stopped running and printed a stack trace whenever an error occurred! Fortunately, it is quite easy to prevent this.

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a `try` block. Immediately following the `try` block, include a `catch` clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a `try` block and a `catch` clause that processes the `ArithmaticException` generated by the division-by-zero error:

```
class Exc2 {
    public static void main(String args[]) {
        int d, a;

        try { // monitor a block of code.
            d = 0;
            a = 42 / d;
            System.out.println("This will not be printed.");
        } catch (ArithmaticException e) { // catch divide-by-zero error
            System.out.println("Division by zero.");
        }
    }
}
```

```
        System.out.println("After catch statement.");
    }
}
```

This program generates the following output:

```
Division by zero.  
After catch statement.
```

Notice that the call to **println()** inside the **try** block is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. Put differently, **catch** is not “called,” so execution never “returns” to the **try** block from a **catch**. Thus, the line “This will not be printed.” is not displayed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try / catch** mechanism.

A **try** and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement. A **catch** statement cannot catch an exception thrown by another **try** statement (except in the case of nested **try** statements, described shortly). The statements that are protected by **try** must be surrounded by curly braces. (That is, they must be within a block.) You cannot use **try** on a single statement.

The goal of most well-constructed **catch** clauses should be to resolve the exceptional condition and then continue on as if the error had never happened. For example, in the next program each iteration of the **for** loop obtains two random integers. Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into **a**. If either division operation causes a divide-by-zero error, it is caught, the value of **a** is set to zero, and the program continues.

```
// Handle an exception and move on.
import java.util.Random;

class HandleError {
    public static void main(String args[]) {
        int a=0, b=0, c=0;
        Random r = new Random();

        for(int i=0; i<32000; i++) {
            try {
                b = r.nextInt();
                c = r.nextInt();
                a = 12345 / (b/c);
            } catch (ArithmaticException e) {
                System.out.println("Division by zero.");
                a = 0; // set a to zero and continue
            }
            System.out.println("a: " + a);
        }
    }
}
```

Displaying a Description of an Exception

Throwable overrides the `toString()` method (defined by **Object**) so that it returns a string containing a description of the exception. You can display this description in a `println()` statement by simply passing the exception as an argument. For example, the `catch` block in the preceding program can be rewritten like this:

```
catch (ArithmecticException e) {
    System.out.println("Exception: " + e);
    a = 0; // set a to zero and continue
}
```

When this version is substituted in the program, and the program is run, each divide-by-zero error displays the following message:

```
Exception: java.lang.ArithmecticException: / by zero
```

While it is of no particular value in this context, the ability to display a description of an exception is valuable in other circumstances—particularly when you are experimenting with exceptions or when you are debugging.

Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more `catch` clauses, each catching a different type of exception. When an exception is thrown, each `catch` statement is inspected in order, and the first one whose type matches that of the exception is executed. After one `catch` statement executes, the others are bypassed, and execution continues after the `try / catch` block. The following example traps two different exception types:

```
// Demonstrate multiple catch statements.
class MultipleCatches {
    public static void main(String args[]) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[42] = 99;
        } catch(ArithmecticException e) {
            System.out.println("Divide by 0: " + e);
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index oob: " + e);
        }
        System.out.println("After try/catch blocks.");
    }
}
```

This program will cause a division-by-zero exception if it is started with no command-line arguments, since `a` will equal zero. It will survive the division if you provide a command-

line argument, setting **a** to something larger than zero. But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.

Here is the output generated by running it both ways:

```
C:\>java MultipleCatches
a = 0
Divide by 0: java.lang.ArithmetricException: / by zero
After try/catch blocks.

C:\>java MultipleCatches TestArg
a = 1
Array index oob:  java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error. For example, consider the following program:

```
/* This program contains an error.

A subclass must come before its superclass in
a series of catch statements. If not,
unreachable code will be created and a
compile-time error will result.

*/
class SuperSubCatch {
    public static void main(String args[]) {
        try {
            int a = 0;
            int b = 42 / a;
        } catch(Exception e) {
            System.out.println("Generic Exception catch.");
        }
        /* This catch is never reached because
           ArithmeticException is a subclass of Exception. */
        catch(ArithmetricException e) { // ERROR - unreachable
            System.out.println("This is never reached.");
        }
    }
}
```

If you try to compile this program, you will receive an error message stating that the second **catch** statement is unreachable because the exception has already been caught. Since **ArithmetricException** is a subclass of **Exception**, the first **catch** statement will handle all **Exception**-based errors, including **ArithmetricException**. This means that the second **catch** statement will never execute. To fix the problem, reverse the order of the **catch** statements.

Nested try Statements

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**. Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted. If no **catch** statement matches, then the Java run-time system will handle the exception. Here is an example that uses nested **try** statements:

```
// An example of nested try statements.
class NestTry {
    public static void main(String args[]) {
        try {
            int a = args.length;

            /* If no command-line args are present,
               the following statement will generate
               a divide-by-zero exception. */
            int b = 42 / a;

            System.out.println("a = " + a);

            try { // nested try block
                /* If one command-line arg is used,
                   then a divide-by-zero exception
                   will be generated by the following code. */
                if(a==1) a = a/(a-a); // division by zero

                /* If two command-line args are used,
                   then generate an out-of-bounds exception. */
                if(a==2) {
                    int c[] = { 1 };
                    c[42] = 99; // generate an out-of-bounds exception
                }
            } catch(ArrayIndexOutOfBoundsException e) {
                System.out.println("Array index out-of-bounds: " + e);
            }

        } catch(ArithmaticException e) {
            System.out.println("Divide by 0: " + e);
        }
    }
}
```

As you can see, this program nests one **try** block within another. The program works as follows. When you execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer **try** block. Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested **try** block. Since the inner block does not catch this exception, it is passed on to the outer **try** block, where it is handled. If you execute the program with two command-line arguments,

an array boundary exception is generated from within the inner **try** block. Here are sample runs that illustrate each case:

```
C:\>java NestTry
Divide by 0: java.lang.ArithmaticException: / by zero

C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmaticException: / by zero

C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
    java.lang.ArrayIndexOutOfBoundsException:42
```

Nesting of **try** statements can occur in less obvious ways when method calls are involved. For example, you can enclose a call to a method within a **try** block. Inside that method is another **try** statement. In this case, the **try** within the method is still nested inside the outer **try** block, which calls the method. Here is the previous program recoded so that the nested **try** block is moved inside the method **nesttry()**:

```
/* Try statements can be implicitly nested via
   calls to methods. */
class MethNestTry {
    static void nesttry(int a) {
        try { // nested try block
            /* If one command-line arg is used,
               then a divide-by-zero exception
               will be generated by the following code. */
            if(a==1) a = a/(a-a); // division by zero

            /* If two command-line args are used,
               then generate an out-of-bounds exception. */
            if(a==2) {
                int c[] = { 1 };
                c[42] = 99; // generate an out-of-bounds exception
            }
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out-of-bounds: " + e);
        }
    }

    public static void main(String args[]) {
        try {
            int a = args.length;

            /* If no command-line args are present,
               the following statement will generate
               a divide-by-zero exception. */
            int b = 42 / a;
            System.out.println("a = " + a);
        }
    }
}
```

```
    nesttry(a);
} catch(ArithmaticException e) {
    System.out.println("Divide by 0: " + e);
}
}
```

The output of this program is identical to that of the preceding example.

throw

So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

throw *ThrowableInstance*;

Here, `ThrowableInstance` must be an object of type `Throwable` or a subclass of `Throwable`. Primitive types, such as `int` or `char`, as well as non-`Throwable` classes, such as `String` and `Object`, cannot be used as exceptions. There are two ways you can obtain a `Throwable` object: using a parameter in a `catch` clause or creating one with the `new` operator.

The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
// Demonstrate throw.
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch(NullPointerException e) {
            System.out.println("Caught inside demoproc.");
            throw e; // rethrow the exception
        }
    }

    public static void main(String args[]) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            System.out.println("Recaught: " + e);
        }
    }
}
```

This program gets two chances to deal with the same error. First, `main()` sets up an exception context and then calls `demoproc()`. The `demoproc()` method then sets up

another exception-handling context and immediately throws a new instance of **NullPointerException**, which is caught on the next line. The exception is then rethrown. Here is the resulting output:

```
Caught inside demoproc.  
Recaught: java.lang.NullPointerException: demo
```

The program also illustrates how to create one of Java's standard exception objects. Pay close attention to this line:

```
throw new NullPointerException("demo");
```

Here, **new** is used to construct an instance of **NullPointerException**. Many of Java's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print()** or **println()**. It can also be obtained by a call to **getMessage()**, which is defined by **Throwable**.

throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

This is the general form of a method declaration that includes a **throws** clause:

```
type method-name(parameter-list) throws exception-list  
{  
    // body of method  
}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a **throws** clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.  
class ThrowsDemo {  
    static void throwOne() {  
        System.out.println("Inside throwOne.");  
        throw new IllegalAccessException("demo");  
    }  
    public static void main(String args[]) {  
        throwOne();  
    }  
}
```

To make this example compile, you need to make two changes. First, you need to declare that `throwOne()` throws `IllegalAccessException`. Second, `main()` must define a `try / catch` statement that catches this exception.

The corrected example is shown here:

```
// This is now correct.
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        try {
            throwOne();
        } catch (IllegalAccessException e) {
            System.out.println("Caught " + e);
        }
    }
}
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

finally

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The `finally` keyword is designed to address this contingency.

`finally` creates a block of code that will be executed after a `try /catch` block has completed and before the code following the `try/catch` block. The `finally` block will execute whether or not an exception is thrown. If an exception is thrown, the `finally` block will execute even if no `catch` statement matches the exception. Any time a method is about to return to the caller from inside a `try/catch` block, via an uncaught exception or an explicit return statement, the `finally` clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The `finally` clause is optional. However, each `try` statement requires at least one `catch` or a `finally` clause.

Here is an example program that shows three methods that exit in various ways, none without executing their **finally** clauses:

```
// Demonstrate finally.
class FinallyDemo {
    // Throw an exception out of the method.
    static void procA() {
        try {
            System.out.println("inside procA");
            throw new RuntimeException("demo");
        } finally {
            System.out.println("procA's finally");
        }
    }

    // Return from within a try block.
    static void procB() {
        try {
            System.out.println("inside procB");
            return;
        } finally {
            System.out.println("procB's finally");
        }
    }

    // Execute a try block normally.
    static void procC() {
        try {
            System.out.println("inside procC");
        } finally {
            System.out.println("procC's finally");
        }
    }

    public static void main(String args[]) {
        try {
            procA();
        } catch (Exception e) {
            System.out.println("Exception caught");
        }

        procB();
        procC();
    }
}
```

In this example, **procA()** prematurely breaks out of the **try** by throwing an exception. The **finally** clause is executed on the way out. **procB()**'s **try** statement is exited via a **return** statement. The **finally** clause is executed before **procB()** returns. In **procC()**, the **try** statement executes normally, without error. However, the **finally** block is still executed.

REMEMBER If a **finally** block is associated with a **try**, the **finally** block will be executed upon conclusion of the **try**.

Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

Java's Built-in Exceptions

Inside the standard package `java.lang`, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type `RuntimeException`. As previously explained, these exceptions need not be included in any method's `throws` list. In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in `java.lang` are listed in Table 10-1. Table 10-2 lists those exceptions defined by `java.lang` that must be included in a method's `throws` list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions*. In addition to the exceptions in `java.lang`, Java defines several more that relate to its other standard packages.

Exception	Meaning
<code>ArithmaticException</code>	Arithmetic error, such as divide-by-zero.
<code>ArrayIndexOutOfBoundsException</code>	Array index is out-of-bounds.
<code>ArrayStoreException</code>	Assignment to an array element of an incompatible type.
<code>ClassCastException</code>	Invalid cast.
<code>EnumConstantNotPresentException</code>	An attempt is made to use an undefined enumeration value.
<code>IllegalArgumentException</code>	Illegal argument used to invoke a method.
<code>IllegalMonitorStateException</code>	Illegal monitor operation, such as waiting on an unlocked thread.
<code>IllegalStateException</code>	Environment or application is in incorrect state.
<code>IllegalThreadStateException</code>	Requested operation not compatible with current thread state.
<code>IndexOutOfBoundsException</code>	Some type of index is out-of-bounds.
<code>NegativeArraySizeException</code>	Array created with a negative size.
<code>NullPointerException</code>	Invalid use of a null reference.
<code>NumberFormatException</code>	Invalid conversion of a string to a numeric format.
<code>SecurityException</code>	Attempt to violate security.
<code>StringIndexOutOfBoundsException</code>	Attempt to index outside the bounds of a string.
<code>TypeNotPresentException</code>	Type not found.
<code>UnsupportedOperationException</code>	An unsupported operation was encountered.

Table 10-1 Java's Unchecked `RuntimeException` Subclasses Defined in `java.lang`

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

Table 10-2 Java's Checked Exceptions Defined in `java.lang`

Creating Your Own Exception Subclasses

Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications. This is quite easy to do: just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**). Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.

The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them. They are shown in Table 10-3. You may also wish to override one or more of these methods in exception classes that you create.

Exception defines four public constructors. Two support chained exceptions, described in the next section. The other two are shown here:

```
Exception( )
Exception(String msg)
```

The first form creates an exception that has no description. The second form lets you specify a description of the exception.

Although specifying a description when an exception is created is often useful, sometimes it is better to override **toString()**. Here's why: The version of **toString()** defined by **Throwable** (and inherited by **Exception**) first displays the name of the exception followed by a colon, which is then followed by your description. By overriding **toString()**, you can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.

Method	Description
final void addSuppressed(Throwable <i>exc</i>)	Adds <i>exc</i> to the list of suppressed exceptions associated with the invoking exception. Primarily for use by the try-with-resources statement.
Throwable fillInStackTrace()	Returns a Throwable object that contains a completed stack trace. This object can be rethrown.
Throwable getCause()	Returns the exception that underlies the current exception. If there is no underlying exception, null is returned.
String getLocalizedMessage()	Returns a localized description of the exception.
String getMessage()	Returns a description of the exception.
StackTraceElement[] getStackTrace()	Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement . The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The StackTraceElement class gives your program access to information about each element in the trace, such as its method name.
final Throwable[] getSuppressed()	Obtains the suppressed exceptions associated with the invoking exception and returns an array that contains the result. Suppressed exceptions are primarily generated by the try-with-resources statement.
Throwable initCause(Throwable <i>causeExc</i>)	Associates <i>causeExc</i> with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace()	Displays the stack trace.
void printStackTrace(PrintStream <i>stream</i>)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter <i>stream</i>)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement <i>elements</i> [])	Sets the stack trace to the elements passed in <i>elements</i> . This method is for specialized applications, not normal use.
String toString()	Returns a String object containing a description of the exception. This method is called by println() when outputting a Throwable object.

Table 10-3 The Methods Defined by **Throwable**

The following example declares a new subclass of **Exception** and then uses that subclass to signal an error condition in a method. It overrides the **toString()** method, allowing a carefully tailored description of the exception to be displayed.

```
// This program creates a custom exception type.
class MyException extends Exception {
    private int detail;

    MyException(int a) {
        detail = a;
    }

    public String toString() {
        return "MyException[" + detail + "]";
    }
}

class ExceptionDemo {
    static void compute(int a) throws MyException {
        System.out.println("Called compute(" + a + ")");
        if(a > 10)
            throw new MyException(a);
        System.out.println("Normal exit");
    }

    public static void main(String args[]) {
        try {
            compute(1);
            compute(20);
        } catch (MyException e) {
            System.out.println("Caught " + e);
        }
    }
}
```

This example defines a subclass of **Exception** called **MyException**. This subclass is quite simple: It has only a constructor plus an overridden **toString()** method that displays the value of the exception. The **ExceptionDemo** class defines a method named **compute()** that throws a **MyException** object. The exception is thrown when **compute()**'s integer parameter is greater than 10. The **main()** method sets up an exception handler for **MyException**, then calls **compute()** with a legal value (less than 10) and an illegal one to show both paths through the code. Here is the result:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

Chained Exceptions

Beginning with JDK 1.4, a feature was incorporated into the exception subsystem: *chained exceptions*. The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception. For example, imagine a situation in which a method throws an **ArithmaticException** because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly. Although the method must certainly throw an **ArithmaticException**, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error. Chained exceptions let you handle this, and any other situation in which layers of exceptions exist.

To allow chained exceptions, two constructors and two methods were added to **Throwable**. The constructors are shown here:

```
Throwable(Throwable causeExc)
Throwable(String msg, Throwable causeExc)
```

In the first form, *causeExc* is the exception that causes the current exception. That is, *causeExc* is the underlying reason that an exception occurred. The second form allows you to specify a description at the same time that you specify a cause exception. These two constructors have also been added to the **Error**, **Exception**, and **RuntimeException** classes.

The chained exception methods supported by **Throwable** are **getCause()** and **initCause()**. These methods are shown in Table 10-3 and are repeated here for the sake of discussion.

```
Throwable getCause()
Throwable initCause(Throwable causeExc)
```

The **getCause()** method returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned. The **initCause()** method associates *causeExc* with the invoking exception and returns a reference to the exception. Thus, you can associate a cause with an exception after the exception has been created. However, the cause exception can be set only once. Thus, you can call **initCause()** only once for each exception object. Furthermore, if the cause exception was set by a constructor, then you can't set it again using **initCause()**. In general, **initCause()** is used to set a cause for legacy exception classes that don't support the two additional constructors described earlier.

Here is an example that illustrates the mechanics of handling chained exceptions:

```
// Demonstrate exception chaining.
class ChainExcDemo {
    static void demoproc() {

        // create an exception
        NullPointerException e =
            new NullPointerException("top layer");

        // add a cause
        e.initCause(new ArithmaticException("cause"));

        throw e;
    }
}
```

```

public static void main(String args[]) {
    try {
        demoproc();
    } catch(NullPointerException e) {
        // display top level exception
        System.out.println("Caught: " + e);

        // display cause exception
        System.out.println("Original cause: " +
                           e.getCause());
    }
}
}

```

The output from the program is shown here:

```

Caught: java.lang.NullPointerException: top layer
Original cause: java.lang.ArithmetricException: cause

```

In this example, the top-level exception is **NullPointerException**. To it is added a cause exception, **ArithmetricException**. When the exception is thrown out of **demoproc()**, it is caught by **main()**. There, the top-level exception is displayed, followed by the underlying exception, which is obtained by calling **getCause()**.

Chained exceptions can be carried on to whatever depth is necessary. Thus, the cause exception can, itself, have a cause. Be aware that overly long chains of exceptions may indicate poor design.

Chained exceptions are not something that every program will need. However, in cases in which knowledge of an underlying cause is useful, they offer an elegant solution.

Three Recently Added Exception Features

Beginning with JDK 7, three interesting and useful features have been added to the exception system. The first automates the process of releasing a resource, such as a file, when it is no longer needed. It is based on an expanded form of the **try** statement called **try-with-resources**, and is described in Chapter 13 when files are introduced. The second feature is called *multi-catch*, and the third is sometimes referred to as *final rethrow* or *more precise rethrow*. These two features are described here.

The multi-catch feature allows two or more exceptions to be caught by the same **catch** clause. It is not uncommon for two or more exception handlers to use the same code sequence even though they respond to different exceptions. Instead of having to catch each exception type individually, you can use a single **catch** clause to handle all of the exceptions without code duplication.

To use a multi-catch, separate each exception type in the **catch** clause with the OR operator. Each multi-catch parameter is implicitly **final**. (You can explicitly specify **final**, if desired, but it is not necessary.) Because each multi-catch parameter is implicitly **final**, it can't be assigned a new value.

Here is a **catch** statement that uses the multi-catch feature to catch both **ArithmaticException** and **ArrayIndexOutOfBoundsException**:

```
catch(ArithmaticException | ArrayIndexOutOfBoundsException e) {
```

The following program shows the multi-catch feature in action:

```
// Demonstrate the multi-catch feature.
class MultiCatch {
    public static void main(String args[]) {
        int a=10, b=0;
        int vals[] = { 1, 2, 3 };

        try {
            int result = a / b; // generate an ArithmaticException
//          vals[10] = 19; // generate an ArrayIndexOutOfBoundsException

            // This catch clause catches both exceptions.
        } catch(ArithmaticException | ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception caught: " + e);
        }

        System.out.println("After multi-catch.");
    }
}
```

The program will generate an **ArithmaticException** when the division by zero is attempted. If you comment out the division statement and remove the comment symbol from the next line, an **ArrayIndexOutOfBoundsException** is generated. Both exceptions are caught by the single **catch** statement.

The more precise rethrow feature restricts the type of exceptions that can be rethrown to only those checked exceptions that the associated **try** block throws, that are not handled by a preceding **catch** clause, and that are a subtype or supertype of the parameter. Although this capability might not be needed often, it is now available for use. For the more precise rethrow feature to be in force, the **catch** parameter must be either effectively **final**, which means that it must not be assigned a new value inside the **catch** block, or explicitly declared **final**.

Using Exceptions

Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics. It is important to think of **try**, **throw**, and **catch** as clean ways to handle errors and unusual boundary conditions in your program's logic. Unlike some other languages in which error return codes are used to indicate failure, Java uses exceptions. Thus, when a method can fail, have it throw an exception. This is a cleaner way to handle failure modes.

One last point: Java's exception-handling statements should not be considered a general mechanism for nonlocal branching. If you do so, it will only confuse your code and make it hard to maintain.

CHAPTER

11

Multithreaded Programming

Java provides built-in support for *multithreaded programming*. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread*, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

You are almost certainly acquainted with multitasking because it is supported by virtually all modern operating systems. However, there are two distinct types of multitasking: process-based and thread-based. It is important to understand the difference between the two. For many readers, process-based multitasking is the more familiar form. A *process* is, in essence, a program that is executing. Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a web site. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details.

Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly. Threads, on the other hand, are lighter weight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost. While Java programs make use of process-based multitasking environments, process-based multitasking is not under Java’s control. However, multithreaded multitasking is.

Multithreading enables you to write efficient programs that make maximum use of the processing power available in the system. One important way multithreading achieves this is by keeping idle time to a minimum. This is especially important for the interactive, networked

environment in which Java operates because idle time is common. For example, the transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a much slower pace than they can be processed by the CPU. And, of course, user input is much slower than the computer. In a single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one—even though most of the time the program is idle, waiting for input. Multithreading helps you reduce this idle time because another thread can run when one is waiting.

If you have programmed for operating systems such as Windows, then you are already familiar with multithreaded programming. However, the fact that Java manages threads makes multithreading especially convenient because many of the details are handled for you.

The Java Thread Model

The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.

The value of a multithreaded environment is best understood in contrast to its counterpart. Single-threaded systems use an approach called an *event loop* with *polling*. In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next. Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the program. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed. In general, in a single-threaded environment, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, the entire program stops running.

The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program. For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

As most readers know, over the past few years, multi-core systems have become commonplace. Of course, single-core systems are still in widespread use. It is important to understand that Java's multithreading features work in both types of systems. In a single-core system, concurrently executing threads share the CPU, with each thread receiving a slice of CPU time. Therefore, in a single-core system, two or more threads do not actually run at the same time, but idle CPU time is utilized. However, in multi-core systems, it is possible for two or more threads to actually execute simultaneously. In many cases, this can further improve program efficiency and increase the speed of certain operations.

NOTE Recently, the Fork/Join Framework was added to Java. It provides a powerful means of creating multithreaded applications that automatically scale to make best use of multi-core environments. The Fork/Join Framework is part of Java's support for *parallel programming*, which is the name commonly given to the techniques that optimize some types of algorithms for parallel execution in systems that have more than one CPU. For a discussion of the Fork/Join Framework and other concurrency utilities, see Chapter 28. Java's traditional multithreading capabilities are described here.

Threads exist in several states. Here is a general description. A thread can be *running*. It can be *ready to run* as soon as it gets CPU time. A running thread can be *suspended*, which temporarily halts its activity. A suspended thread can then be *resumed*, allowing it to pick up where it left off. A thread can be *blocked* when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch*. The rules that determine when a context switch takes place are simple:

- A *thread can voluntarily relinquish control*. This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
- A *thread can be preempted by a higher-priority thread*. In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing—by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking*.

In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems such as Windows, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

CAUTION Portability problems can arise from the differences in the way that operating systems context-switch threads of equal priority.

Synchronization

Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it. For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you

need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the *monitor*. The monitor is a control mechanism first defined by C.A.R. Hoare. You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

In Java, there is no class "Monitor"; instead, each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. This enables you to write very clear and concise multithreaded code, because synchronization support is built into the language.

Messaging

After you divide your program into separate threads, you need to define how they will communicate with each other. When programming with some other languages, you must depend on the operating system to establish communication between threads. This, of course, adds overhead. By contrast, Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have. Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

The Thread Class and the Runnable Interface

Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. **Thread** encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the **Thread** instance that spawned it. To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.

The **Thread** class defines several methods that help manage threads. Several of those used in this chapter are shown here:

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

Thus far, all the examples in this book have used a single thread of execution. The remainder of this chapter explains how to use **Thread** and **Runnable** to create and manage threads, beginning with the one thread that all Java programs have: the main thread.

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other “child” threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread()**, which is a **public static** member of **Thread**. Its general form is shown here:

```
static Thread currentThread()
```

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Let's begin by reviewing the following example:

```
// Controlling the main Thread.
class CurrentThreadDemo {
    public static void main(String args[]) {
        Thread t = Thread.currentThread();

        System.out.println("Current thread: " + t);

        // change the name of the thread
        t.setName("My Thread");
        System.out.println("After name change: " + t);

        try {
            for(int n = 5; n > 0; n--) {
                System.out.println(n);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }
    }
}
```

In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread()**, and this reference is stored in the local variable **t**. Next, the program displays information about the thread. The program then calls **setName()** to change the internal name of the thread. Information about the thread is then redisplayed. Next, a loop counts down from five, pausing one second between each line. The pause is accomplished by the **sleep()** method. The argument to **sleep()** specifies the delay period in milliseconds. Notice the **try/catch** block around this loop. The **sleep()** method in **Thread** might throw an **InterruptedException**. This would happen if some other

thread wanted to interrupt this sleeping one. This example just prints a message if it gets interrupted. In a real program, you would need to handle this differently. Here is the output generated by this program:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Notice the output produced when `t` is used as an argument to `println()`. This displays, in order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is `main`. Its priority is 5, which is the default value, and `main` is also the name of the group of threads to which this thread belongs. A *thread group* is a data structure that controls the state of a collection of threads as a whole. After the name of the thread is changed, `t` is again output. This time, the new name of the thread is displayed.

Let's look more closely at the methods defined by `Thread` that are used in the program. The `sleep()` method causes the thread from which it is called to suspend execution for the specified period of milliseconds. Its general form is shown here:

```
static void sleep(long milliseconds) throws InterruptedException
```

The number of milliseconds to suspend is specified in `milliseconds`. This method may throw an `InterruptedException`.

The `sleep()` method has a second form, shown next, which allows you to specify the period in terms of milliseconds and nanoseconds:

```
static void sleep(long milliseconds, int nanoseconds) throws InterruptedException
```

This second form is useful only in environments that allow timing periods as short as nanoseconds.

As the preceding program shows, you can set the name of a thread by using `setName()`. You can obtain the name of a thread by calling `getName()` (but note that this is not shown in the program). These methods are members of the `Thread` class and are declared like this:

```
final void setName(String threadName)
final String getName()
```

Here, `threadName` specifies the name of the thread.

Creating a Thread

In the most general sense, you create a thread by instantiating an object of type `Thread`. Java defines two ways in which this can be accomplished:

- You can implement the `Runnable` interface.
- You can extend the `Thread` class, itself.

The following two sections look at each method, in turn.

Implementing Runnable

The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called **run()**, which is declared like this:

```
public void run()
```

Inside **run()**, you will define the code that constitutes the new thread. It is important to understand that **run()** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run()** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run()** returns.

After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we will use is shown here:

```
Thread(Runnable threadOb, String threadName)
```

In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*.

After the new thread is created, it will not start running until you call its **start()** method, which is declared within **Thread**. In essence, **start()** executes a call to **run()**. The **start()** method is shown here:

```
void start()
```

Here is an example that creates a new thread and starts it running:

```
// Create a second thread.
class NewThread implements Runnable {
    Thread t;

    NewThread() {
        // Create a new, second thread
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + t);
        t.start(); // Start the thread
    }

    // This is the entry point for the second thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}
```

```

}

class ThreadDemo {
    public static void main(String args[ ] ) {
        new NewThread(); // create a new thread

        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {

            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}

```

Inside **NewThread**'s constructor, a new **Thread** object is created by the following statement:

```
t = new Thread(this, "Demo Thread");
```

Passing **this** as the first argument indicates that you want the new thread to call the **run()** method on **this** object. Next, **start()** is called, which starts the thread of execution beginning at the **run()** method. This causes the child thread's **for** loop to begin. After calling **start()**, **NewThread**'s constructor returns to **main()**. When the main thread resumes, it enters its **for** loop. Both threads continue running, sharing the CPU in single-core systems, until their loops finish. The output produced by this program is as follows. (Your output may vary based upon the specific execution environment.)

```

Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.

```

As mentioned earlier, in a multithreaded program, often the main thread must be the last thread to finish running. In fact, for some older JVMs, if the main thread finishes before a child thread has completed, then the Java run-time system may “hang.” The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to terminate earlier than the main thread. Shortly, you will see a better way to wait for a thread to finish.

Extending Thread

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class. The extending class must override the **run()** method, which is the entry point for the new thread. It must also call **start()** to begin execution of the new thread. Here is the preceding program rewritten to extend **Thread**:

```
// Create a second thread by extending Thread
class NewThread extends Thread {

    NewThread() {
        // Create a new, second thread
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start(); // Start the thread
    }

    // This is the entry point for the second thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

class ExtendThread {
    public static void main(String args[]) {
        new NewThread(); // create a new thread

        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**.

Notice the call to `super()` inside `NewThread`. This invokes the following form of the `Thread` constructor:

```
public Thread(String threadName)
```

Here, `threadName` specifies the name of the thread.

Choosing an Approach

At this point, you might be wondering why Java has two ways to create child threads, and which approach is better. The answers to these questions turn on the same point. The `Thread` class defines several methods that can be overridden by a derived class. Of these methods, the only one that *must* be overridden is `run()`. This is, of course, the same method required when you implement `Runnable`. Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if you will not be overriding any of `Thread`'s other methods, it is probably best simply to implement `Runnable`. Also, by implementing `Runnable`, your thread class does not need to inherit `Thread`, making it free to inherit a different class. Ultimately, which approach to use is up to you. However, throughout the rest of this chapter, we will create threads by using classes that implement `Runnable`.

Creating Multiple Threads

So far, you have been using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs. For example, the following program creates three child threads:

```
// Create multiple threads.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " Interrupted");
        }
        System.out.println(name + " exiting.");
    }
}
```

```
class MultiThreadDemo {  
    public static void main(String args[]) {  
        new NewThread("One"); // start threads  
        new NewThread("Two");  
        new NewThread("Three");  
  
        try {  
            // wait for other threads to end  
            Thread.sleep(10000);  
        } catch (InterruptedException e) {  
            System.out.println("Main thread Interrupted");  
        }  
        System.out.println("Main thread exiting.");  
    }  
}
```

Sample output from this program is shown here. (Your output may vary based upon the specific execution environment.)

```
New thread: Thread[One,5,main]  
New thread: Thread[Two,5,main]  
New thread: Thread[Three,5,main]  
One: 5  
Two: 5  
Three: 5  
One: 4  
Two: 4  
Three: 4  
One: 3  
Three: 3  
Two: 3  
One: 2  
Three: 2  
Two: 2  
One: 1  
Three: 1  
Two: 1  
One exiting.  
Two exiting.  
Three exiting.  
Main thread exiting.
```

As you can see, once started, all three child threads share the CPU. Notice the call to **sleep(10000)** in **main()**. This causes the main thread to sleep for ten seconds and ensures that it will finish last.

Using **isAlive()** and **join()**

As mentioned, often you will want the main thread to finish last. In the preceding examples, this is accomplished by calling **sleep()** within **main()**, with a long enough delay to ensure that all child threads terminate prior to the main thread. However, this is hardly a

satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended? Fortunately, **Thread** provides a means by which you can answer this question.

Two ways exist to determine whether a thread has finished. First, you can call **isAlive()** on the thread. This method is defined by **Thread**, and its general form is shown here:

```
final boolean isAlive()
```

The **isAlive()** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.

While **isAlive()** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join()**, shown here:

```
final void join() throws InterruptedException
```

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it. Additional forms of **join()** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Here is an improved version of the preceding example that uses **join()** to ensure that the main thread is the last to stop. It also demonstrates the **isAlive()** method.

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }
}

class DemoJoin {
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");
        NewThread ob3 = new NewThread("Three");
    }
}
```

```
System.out.println("Thread One is alive: "
                  + ob1.t.isAlive());
System.out.println("Thread Two is alive: "
                  + ob2.t.isAlive());
System.out.println("Thread Three is alive: "
                  + ob3.t.isAlive());
// wait for threads to finish
try {
    System.out.println("Waiting for threads to finish.");
    ob1.t.join();
    ob2.t.join();
    ob3.t.join();
} catch (InterruptedException e) {
    System.out.println("Main thread Interrupted");
}

System.out.println("Thread One is alive: "
                  + ob1.t.isAlive());
System.out.println("Thread Two is alive: "
                  + ob2.t.isAlive());
System.out.println("Thread Three is alive: "
                  + ob3.t.isAlive());

System.out.println("Main thread exiting.");
}
}
```

Sample output from this program is shown here. (Your output may vary based upon the specific execution environment.)

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
```

```
One exiting.  
Thread One is alive: false  
Thread Two is alive: false  
Thread Three is alive: false  
Main thread exiting.
```

As you can see, after the calls to `join()` return, the threads have stopped executing.

Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, over a given period of time, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

In theory, threads of equal priority should get equal access to the CPU. But you need to be careful. Remember, Java is designed to work in a wide range of environments. Some of those environments implement multitasking fundamentally differently than others. For safety, threads that share the same priority should yield control once in a while. This ensures that all threads have a chance to run under a nonpreemptive operating system. In practice, even in nonpreemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O. When this happens, the blocked thread is suspended and other threads can run. But, if you want smooth multithreaded execution, you are better off not relying on this. Also, some types of tasks are CPU-intensive. Such threads dominate the CPU. For these types of threads, you want to yield control occasionally so that other threads can run.

To set a thread's priority, use the `setPriority()` method, which is a member of `Thread`. This is its general form:

```
final void setPriority(int level)
```

Here, `level` specifies the new priority setting for the calling thread. The value of `level` must be within the range `MIN_PRIORITY` and `MAX_PRIORITY`. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify `NORM_PRIORITY`, which is currently 5. These priorities are defined as `static final` variables within `Thread`.

You can obtain the current priority setting by calling the `getPriority()` method of `Thread`, shown here:

```
final int getPriority()
```

Implementations of Java may have radically different behavior when it comes to scheduling. Most of the inconsistencies arise when you have threads that are relying on preemptive behavior, instead of cooperatively giving up CPU time. The safest way to obtain predictable, cross-platform behavior with Java is to use threads that voluntarily give up control of the CPU.

Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization*. As you will see, Java provides unique, language-level support for it.

Key to synchronization is the concept of the monitor. A *monitor* is an object that is used as a mutually exclusive lock. Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

You can synchronize your code in either of two ways. Both involve the use of the **synchronized** keyword, and both are examined here.

Using Synchronized Methods

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

To understand the need for synchronization, let's begin with a simple example that does not use it—but should. The following program has three simple classes. The first one, **Callme**, has a single method named **call()**. The **call()** method takes a **String** parameter called **msg**. This method tries to print the **msg** string inside of square brackets. The interesting thing to notice is that after **call()** prints the opening bracket and the **msg** string, it calls **Thread.sleep(1000)**, which pauses the current thread for one second.

The constructor of the next class, **Caller**, takes a reference to an instance of the **Callme** class and a **String**, which are stored in **target** and **msg**, respectively. The constructor also creates a new thread that will call this object's **run()** method. The thread is started immediately. The **run()** method of **Caller** calls the **call()** method on the **target** instance of **Callme**, passing in the **msg** string. Finally, the **Synch** class starts by creating a single instance of **Callme**, and three instances of **Caller**, each with a unique message string. The same instance of **Callme** is passed to each **Caller**.

```
// This program is not synchronized.
class Callme {
    void call(String msg) {
        System.out.print("[" + msg);
        try {
            Thread.sleep(1000);
        } catch(InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}
```

```

}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;

    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }
    public void run() {
        target.call(msg);
    }
}

class Synch {
    public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");

        // wait for threads to end
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch(InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
}

```

Here is the output produced by this program:

```

Hello[Synchronized[World]
]
]
```

As you can see, by calling **sleep()**, the **call()** method allows execution to switch to another thread. This results in the mixed-up output of the three message strings. In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a *race condition*, because the three threads are racing each other to complete the method. This example used **sleep()** to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will occur. This can cause a program to run right one time and wrong the next.

To fix the preceding program, you must *serialize* access to `call()`. That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede `call()`'s definition with the keyword **synchronized**, as shown here:

```
class Callme {
    synchronized void call(String msg) {
        ...
    }
}
```

This prevents other threads from entering `call()` while another thread is using it. After **synchronized** has been added to `call()`, the output of the program is as follows:

```
[Hello]
[Synchronized]
[World]
```

Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the **synchronized** keyword to guard the state from race conditions. Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, nonsynchronized methods on that instance will continue to be callable.

The synchronized Statement

While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.

This is the general form of the **synchronized** statement:

```
synchronized(objRef) {
    // statements to be synchronized
}
```

Here, *objRef* is a reference to the object being synchronized. A synchronized block ensures that a call to a synchronized method that is a member of *objRef*'s class occurs only after the current thread has successfully entered *objRef*'s monitor.

Here is an alternative version of the preceding example, using a synchronized block within the `run()` method:

```
// This program uses a synchronized block.
class Callme {
    void call(String msg) {
```

```

        System.out.print("[" + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;

    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }

    // synchronize calls to call()
    public void run() {
        synchronized(target) { // synchronized block
            target.call(msg);
        }
    }
}

class Synch1 {
    public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");

        // wait for threads to end
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch(InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
}

```

Here, the **call()** method is not modified by **synchronized**. Instead, the **synchronized** statement is used inside **Caller**'s **run()** method. This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

Interthread Communication

The preceding examples unconditionally blocked other threads from asynchronous access to certain methods. This use of the implicit monitors in Java objects is powerful, but you can achieve a more subtle level of control through interprocess communication. As you will see, this is especially easy in Java.

As discussed earlier, multithreading replaces event loop programming by dividing your tasks into discrete, logical units. Threads also provide a secondary benefit: they do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time. For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.

To avoid polling, Java includes an elegant interprocess communication mechanism via the `wait()`, `notify()`, and `notifyAll()` methods. These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context. Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:

- `wait()` tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls `notify()` or `notifyAll()`.
- `notify()` wakes up a thread that called `wait()` on the same object.
- `notifyAll()` wakes up all the threads that called `wait()` on the same object. One of the threads will be granted access.

These methods are declared within **Object**, as shown here:

```
final void wait() throws InterruptedException  
final void notify()  
final void notifyAll()
```

Additional forms of `wait()` exist that allow you to specify a period of time to wait.

Before working through an example that illustrates interthread communication, an important point needs to be made. Although `wait()` normally waits until `notify()` or `notifyAll()` is called, there is a possibility that in very rare cases the waiting thread could be awakened due to a *spurious wakeup*. In this case, a waiting thread resumes without `notify()` or `notifyAll()` having been called. (In essence, the thread resumes for no apparent reason.) Because of this remote possibility, Oracle recommends that calls to `wait()` should take place within a loop that checks the condition on which the thread is waiting. The following example shows this technique.

Let's now work through an example that uses `wait()` and `notify()`. To begin, consider the following sample program that incorrectly implements a simple form of the producer/consumer problem. It consists of four classes: **Q**, the queue that you're trying to synchronize; **Producer**, the threaded object that is producing queue entries; **Consumer**, the threaded

object that is consuming queue entries; and **PC**, the tiny class that creates the single **Q**, **Producer**, and **Consumer**.

```
// An incorrect implementation of a producer and consumer.
class Q {
    int n;

    synchronized int get() {
        System.out.println("Got: " + n);
        return n;
    }

    synchronized void put(int n) {
        this.n = n;
        System.out.println("Put: " + n);
    }
}

class Producer implements Runnable {
    Q q;

    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    public void run() {
        int i = 0;

        while(true) {
            q.put(i++);
        }
    }
}

class Consumer implements Runnable {
    Q q;

    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }

    public void run() {
        while(true) {
            q.get();
        }
    }
}

class PC {
    public static void main(String args[]) {
```

```

    Q q = new Q();
    new Producer(q);
    new Consumer(q);

    System.out.println("Press Control-C to stop.");
}
}

```

Although the **put()** and **get()** methods on **Q** are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue value twice. Thus, you get the erroneous output shown here (the exact output will vary with processor speed and task load):

```

Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7

```

As you can see, after the producer put 1, the consumer started and got the same 1 five times in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer have a chance to consume them.

The proper way to write this program in Java is to use **wait()** and **notify()** to signal in both directions, as shown here:

```

// A correct implementation of a producer and consumer.
class Q {
    int n;
    boolean valueSet = false;

    synchronized int get() {
        while(!valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }

        System.out.println("Got: " + n);
        valueSet = true;
        notify();
        return n;
    }

    synchronized void put(int n) {

```

```
while(valueSet)
    try {
        wait();
    } catch(InterruptedException e) {
        System.out.println("InterruptedException caught");
    }

    this.n = n;
    valueSet = true;
    System.out.println("Put: " + n);
    notify();
}
}

class Producer implements Runnable {
    Q q;

    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    public void run() {
        int i = 0;

        while(true) {
            q.put(i++);
        }
    }
}

class Consumer implements Runnable {
    Q q;

    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }

    public void run() {
        while(true) {
            q.get();
        }
    }
}

class PCFixed {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);

        System.out.println("Press Control-C to stop.");
    }
}
```

Inside `get()`, `wait()` is called. This causes its execution to suspend until **Producer** notifies you that some data is ready. When this happens, execution inside `get()` resumes. After the data has been obtained, `get()` calls `notify()`. This tells **Producer** that it is okay to put more data in the queue. Inside `put()`, `wait()` suspends execution until **Consumer** has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and `notify()` is called. This tells **Consumer** that it should now remove it.

Here is some output from this program, which shows the clean synchronous behavior:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

Deadlock

A special type of error that you need to avoid that relates specifically to multitasking is *deadlock*, which occurs when two threads have a circular dependency on a pair of synchronized objects. For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete. Deadlock is a difficult error to debug for two reasons:

- In general, it occurs only rarely, when the two threads time-slice in just the right way.
- It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)

To understand deadlock fully, it is useful to see it in action. The next example creates two classes, **A** and **B**, with methods `foo()` and `bar()`, respectively, which pause briefly before trying to call a method in the other class. The main class, named **Deadlock**, creates an **A** and a **B** instance, and then starts a second thread to set up the deadlock condition. The `foo()` and `bar()` methods use `sleep()` as a way to force the deadlock condition to occur.

```
// An example of deadlock.
class A {
    synchronized void foo(B b) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered A.foo");
        try {
            Thread.sleep(1000);
        } catch(Exception e) {
            System.out.println("A Interrupted");
        }
    }
}
```

```
}

    System.out.println(name + " trying to call B.last()");
    b.last();
}

synchronized void last() {
    System.out.println("Inside A.last");
}
}

class B {
    synchronized void bar(A a) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered B.bar");

        try {
            Thread.sleep(1000);
        } catch(Exception e) {
            System.out.println("B Interrupted");
        }

        System.out.println(name + " trying to call A.last()");
        a.last();
    }

    synchronized void last() {
        System.out.println("Inside A.last");
    }
}

class Deadlock implements Runnable {
    A a = new A();
    B b = new B();

    Deadlock() {
        Thread.currentThread().setName("MainThread");
        Thread t = new Thread(this, "RacingThread");
        t.start();

        a.foo(b); // get lock on a in this thread.
        System.out.println("Back in main thread");
    }

    public void run() {
        b.bar(a); // get lock on b in other thread.
        System.out.println("Back in other thread");
    }
}

public static void main(String args[]) {
    new Deadlock();
}
```

When you run this program, you will see the output shown here:

```
MainThread entered A.foo  
RacingThread entered B.bar  
MainThread trying to call B.last()  
RacingThread trying to call A.last()
```

Because the program has deadlocked, you need to press CTRL-C to end the program. You can see a full thread and monitor cache dump by pressing CTRL-BREAK on a PC. You will see that **RacingThread** owns the monitor on **b**, while it is waiting for the monitor on **a**. At the same time, **MainThread** owns **a** and is waiting to get **b**. This program will never complete. As this example illustrates, if your multithreaded program locks up occasionally, deadlock is one of the first conditions that you should check for.

Suspending, Resuming, and Stopping Threads

Sometimes, suspending execution of a thread is useful. For example, a separate thread can be used to display the time of day. If the user doesn't want a clock, then its thread can be suspended. Whatever the case, suspending a thread is a simple matter. Once suspended, restarting the thread is also a simple matter.

The mechanisms to suspend, stop, and resume threads differ between early versions of Java, such as Java 1.0, and modern versions, beginning with Java 2. Prior to Java 2, a program used **suspend()**, **resume()**, and **stop()**, which are methods defined by **Thread**, to pause, restart, and stop the execution of a thread. Although these methods seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs. Here's why. The **suspend()** method of the **Thread** class was deprecated by Java 2 several years ago. This was done because **suspend()** can sometimes cause serious system failures. Assume that a thread has obtained locks on critical data structures. If that thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked.

The **resume()** method is also deprecated. It does not cause problems, but cannot be used without the **suspend()** method as its counterpart.

The **stop()** method of the **Thread** class, too, was deprecated by Java 2. This was done because this method can sometimes cause serious system failures. Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state. The trouble is that **stop()** causes any lock the calling thread holds to be released. Thus, the corrupted data might be used by another thread that is waiting on the same lock.

Because you can't now use the **suspend()**, **resume()**, or **stop()** methods to control a thread, you might be thinking that no way exists to pause, restart, or terminate a thread. But, fortunately, this is not true. Instead, a thread must be designed so that the **run()** method periodically checks to determine whether that thread should suspend, resume, or stop its own execution. Typically, this is accomplished by establishing a flag variable that indicates the execution state of the thread. As long as this flag is set to "running," the **run()** method must continue to let the thread execute. If this variable is set to "suspend," the thread must pause. If it is set to "stop," the thread must terminate. Of course, a variety of ways exist in which to write such code, but the central theme will be the same for all programs.

The following example illustrates how the **wait()** and **notify()** methods that are inherited from **Object** can be used to control the execution of a thread. Let us consider its operation. The **NewThread** class contains a **boolean** instance variable named **suspendFlag**, which is used to control the execution of the thread. It is initialized to **false** by the constructor. The **run()** method contains a **synchronized** statement block that checks **suspendFlag**. If that variable is **true**, the **wait()** method is invoked to suspend the execution of the thread. The **mysuspend()** method sets **suspendFlag** to **true**. The **myresume()** method sets **suspendFlag** to **false** and invokes **notify()** to wake up the thread. Finally, the **main()** method has been modified to invoke the **mysuspend()** and **myresume()** methods.

```
// Suspending and resuming a thread the modern way.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    boolean suspendFlag;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        suspendFlag = false;
        t.start(); // Start the thread
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 15; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(200);
                synchronized(this) {
                    while(suspendFlag) {
                        wait();
                    }
                }
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }

    synchronized void mysuspend() {
        suspendFlag = true;
    }

    synchronized void myresume() {
        suspendFlag = false;
        notify();
    }
}
```

```

class SuspendResume {
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");

        try {
            Thread.sleep(1000);
            ob1.mysuspend();
            System.out.println("Suspending thread One");
            Thread.sleep(1000);
            ob1.myresume();
            System.out.println("Resuming thread One");
            ob2.mysuspend();
            System.out.println("Suspending thread Two");
            Thread.sleep(1000);
            ob2.myresume();
            System.out.println("Resuming thread Two");
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        // wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish.");
            ob1.t.join();
            ob2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        System.out.println("Main thread exiting.");
    }
}

```

When you run the program, you will see the threads suspend and resume. Later in this book, you will see more examples that use the modern mechanism of thread control. Although this mechanism isn't as "clean" as the old way, nevertheless, it is the way required to ensure that run-time errors don't occur. It is the approach that *must* be used for all new code.

Obtaining A Thread's State

As mentioned earlier in this chapter, a thread can exist in a number of different states. You can obtain the current state of a thread by calling the `getState()` method defined by **Thread**. It is shown here:

```
Thread.State getState()
```

It returns a value of type **Thread.State** that indicates the state of the thread at the time at which the call was made. **State** is an enumeration defined by **Thread**. (An enumeration is a

list of named constants. It is discussed in detail in Chapter 12.) Here are the values that can be returned by `getState()`:

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called <code>sleep()</code> . This state is also entered when a timeout version of <code>wait()</code> or <code>join()</code> is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of <code>wait()</code> or <code>join()</code> .

Figure 11-1 diagrams how the various thread states relate.

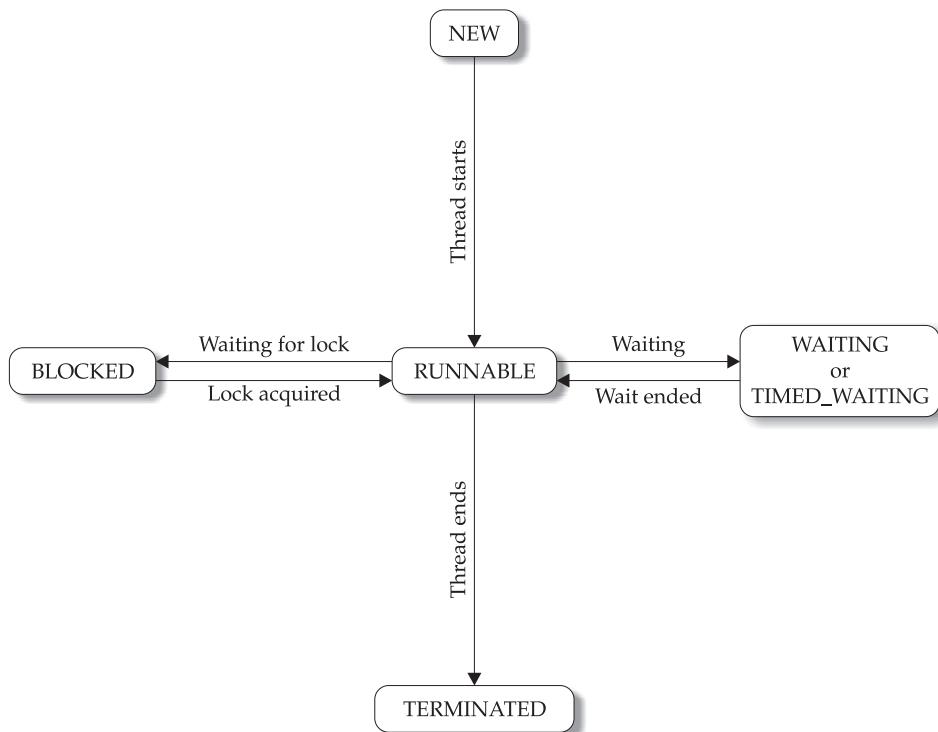


Figure 11-1 Thread states

Given a **Thread** instance, you can use **getState()** to obtain the state of a thread. For example, the following sequence determines if a thread called **thrd** is in the **RUNNABLE** state at the time **getState()** is called:

```
Thread.State ts = thrd.getState();  
if(ts == Thread.State.RUNNABLE) // ...
```

It is important to understand that a thread's state may change after the call to **getState()**. Thus, depending on the circumstances, the state obtained by calling **getState()** may not reflect the actual state of the thread only a moment later. For this (and other) reasons, **getState()** is not intended to provide a means of synchronizing threads. It's primarily used for debugging or for profiling a thread's run-time characteristics.

Using Multithreading

The key to utilizing Java's multithreading features effectively is to think concurrently rather than serially. For example, when you have two subsystems within a program that can execute concurrently, make them individual threads. With the careful use of multithreading, you can create very efficient programs. A word of caution is in order, however: If you create too many threads, you can actually degrade the performance of your program rather than enhance it. Remember, some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program! One last point: To create compute-intensive applications that can automatically scale to make use of the available processors in a multi-core system, consider using the new Fork/Join Framework, which is described in Chapter 28.

This page has been intentionally left blank

CHAPTER

12

Enumerations, Autoboxing, and Annotations (Metadata)

This chapter examines three relatively recent additions to the Java language: enumerations, autoboxing, and annotations (also referred to as metadata). Each expands the power of the language by offering a streamlined approach to handling common programming tasks. This chapter also discusses Java's type wrappers and introduces reflection.

Enumerations

Versions of Java prior to JDK 5 lacked one feature that many programmers felt was needed: enumerations. In its simplest form, an *enumeration* is a list of named constants. Although Java offered other features that provide somewhat similar functionality, such as **final** variables, many programmers still missed the conceptual purity of enumerations—especially because enumerations are supported by many other commonly used languages. Beginning with JDK 5, enumerations were added to the Java language, and they are now an integral and widely used part of Java.

In their simplest form, Java enumerations appear similar to enumerations in other languages. However, this similarity may be only skin deep because, in Java, an enumeration defines a class type. By making enumerations into classes, the capabilities of the enumeration are greatly expanded. For example, in Java, an enumeration can have constructors, methods, and instance variables. Therefore, although enumerations were several years in the making, Java's rich implementation made them well worth the wait.

Enumeration Fundamentals

An enumeration is created using the **enum** keyword. For example, here is a simple enumeration that lists various apple varieties:

```
// An enumeration of apple varieties.  
enum Apple {  
    Jonathan, GoldenDel, RedDel, Winesap, Cortland  
}
```

The identifiers **Jonathan**, **GoldenDel**, and so on, are called *enumeration constants*. Each is implicitly declared as a public, static final member of **Apple**. Furthermore, their type is the type of the enumeration in which they are declared, which is **Apple** in this case. Thus, in the language of Java, these constants are called *self-typed*, in which “self” refers to the enclosing enumeration.

Once you have defined an enumeration, you can create a variable of that type. However, even though enumerations define a class type, you do not instantiate an **enum** using **new**. Instead, you declare and use an enumeration variable in much the same way as you do one of the primitive types. For example, this declares **ap** as a variable of enumeration type **Apple**:

```
Apple ap;
```

Because **ap** is of type **Apple**, the only values that it can be assigned (or can contain) are those defined by the enumeration. For example, this assigns **ap** the value **RedDel**:

```
ap = Apple.RedDel;
```

Notice that the symbol **RedDel** is preceded by **Apple**.

Two enumeration constants can be compared for equality by using the `==` relational operator. For example, this statement compares the value in **ap** with the **GoldenDel** constant:

```
if(ap == Apple.GoldenDel) // ...
```

An enumeration value can also be used to control a **switch** statement. Of course, all of the **case** statements must use constants from the same **enum** as that used by the **switch** expression. For example, this **switch** is perfectly valid:

```
// Use an enum to control a switch statement.
switch(ap) {
    case Jonathan:
        // ...
    case Winesap:
        // ...
```

Notice that in the **case** statements, the names of the enumeration constants are used without being qualified by their enumeration type name. That is, **Winesap**, not **Apple.Winesap**, is used. This is because the type of the enumeration in the **switch** expression has already implicitly specified the **enum** type of the **case** constants. There is no need to qualify the constants in the **case** statements with their **enum** type name. In fact, attempting to do so will cause a compilation error.

When an enumeration constant is displayed, such as in a **println()** statement, its name is output. For example, given this statement:

```
System.out.println(Apple.Winesap);
```

the name **Winesap** is displayed.

The following program puts together all of the pieces and demonstrates the `Apple` enumeration:

```
// An enumeration of apple varieties.  
enum Apple {  
    Jonathan, GoldenDel, RedDel, Winesap, Cortland  
}  
  
class EnumDemo {  
    public static void main(String args[])  
    {  
        Apple ap;  
  
        ap = Apple.RedDel;  
  
        // Output an enum value.  
        System.out.println("Value of ap: " + ap);  
        System.out.println();  
  
        ap = Apple.GoldenDel;  
  
        // Compare two enum values.  
        if(ap == Apple.GoldenDel)  
            System.out.println("ap contains GoldenDel.\n");  
  
        // Use an enum to control a switch statement.  
        switch(ap) {  
            case Jonathan:  
                System.out.println("Jonathan is red.");  
                break;  
            case GoldenDel:  
                System.out.println("Golden Delicious is yellow.");  
                break;  
            case RedDel:  
                System.out.println("Red Delicious is red.");  
                break;  
            case Winesap:  
                System.out.println("Winesap is red.");  
                break;  
            case Cortland:  
                System.out.println("Cortland is red.");  
                break;  
        }  
    }  
}
```

The output from the program is shown here:

```
Value of ap: RedDel  
  
ap contains GoldenDel.  
  
Golden Delicious is yellow.
```

The **values()** and **valueOf()** Methods

All enumerations automatically contain two predefined methods: **values()** and **valueOf()**. Their general forms are shown here:

```
public static enum-type [ ] values( )
public static enum-type valueOf(String str)
```

The **values()** method returns an array that contains a list of the enumeration constants. The **valueOf()** method returns the enumeration constant whose value corresponds to the string passed in *str*. In both cases, *enum-type* is the type of the enumeration. For example, in the case of the **Apple** enumeration shown earlier, the return type of **Apple.valueOf("Winesap")** is **Winesap**.

The following program demonstrates the **values()** and **valueOf()** methods:

```
// Use the built-in enumeration methods.

// An enumeration of apple varieties.
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo2 {
    public static void main(String args[])
    {
        Apple ap;

        System.out.println("Here are all Apple constants:");

        // use values()
        Apple allapples[] = Apple.values();
        for(Apple a : allapples)
            System.out.println(a);

        System.out.println();

        // use valueOf()
        ap = Apple.valueOf("Winesap");
        System.out.println("ap contains " + ap);
    }
}
```

The output from the program is shown here:

```
Here are all Apple constants:
Jonathan
GoldenDel
RedDel
Winesap
Cortland

ap contains Winesap
```

Notice that this program uses a for-each style **for** loop to cycle through the array of constants obtained by calling **values()**. For the sake of illustration, the variable **allapples** was created and assigned a reference to the enumeration array. However, this step is not necessary because the **for** could have been written as shown here, eliminating the need for the **allapples** variable:

```
for(Apple a : Apple.values())
    System.out.println(a);
```

Now, notice how the value corresponding to the name **Winesap** was obtained by calling **valueOf()**.

```
ap = Apple.valueOf("Winesap");
```

As explained, **valueOf()** returns the enumeration value associated with the name of the constant represented as a string.

Java Enumerations Are Class Types

As explained, a Java enumeration is a class type. Although you don't instantiate an **enum** using **new**, it otherwise has much the same capabilities as other classes. The fact that **enum** defines a class gives the Java enumeration extraordinary power. For example, you can give them constructors, add instance variables and methods, and even implement interfaces.

It is important to understand that each enumeration constant is an object of its enumeration type. Thus, when you define a constructor for an **enum**, the constructor is called when each enumeration constant is created. Also, each enumeration constant has its own copy of any instance variables defined by the enumeration. For example, consider the following version of **Apple**:

```
// Use an enum constructor, instance variable, and method.
enum Apple {
    Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

    private int price; // price of each apple

    // Constructor
    Apple(int p) { price = p; }

    int getPrice() { return price; }
}

class EnumDemo3 {
    public static void main(String args[])
    {
        Apple ap;

        // Display price of Winesap.
        System.out.println("Winesap costs " +
                           Apple.Winesap.getPrice() +
                           " cents.\n");

        // Display all apples and prices.
    }
}
```

```

        System.out.println("All apple prices:");
        for(Apple a : Apple.values())
            System.out.println(a + " costs " + a.getPrice() +
                               " cents.");
    }
}

```

The output is shown here:

```
Winesap costs 15 cents.
```

```

All apple prices:
Jonathan costs 10 cents.
GoldenDel costs 9 cents.
RedDel costs 12 cents.
Winesap costs 15 cents.
Cortland costs 8 cents.

```

This version of **Apple** adds three things. The first is the instance variable **price**, which is used to hold the price of each variety of apple. The second is the **Apple** constructor, which is passed the price of an apple. The third is the method **getPrice()**, which returns the value of **price**.

When the variable **ap** is declared in **main()**, the constructor for **Apple** is called once for each constant that is specified. Notice how the arguments to the constructor are specified, by putting them inside parentheses after each constant, as shown here:

```
Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
```

These values are passed to the **p** parameter of **Apple()**, which then assigns this value to **price**. Again, the constructor is called once for each constant.

Because each enumeration constant has its own copy of **price**, you can obtain the price of a specified type of apple by calling **getPrice()**. For example, in **main()** the price of a Winesap is obtained by the following call:

```
Apple.Winesap.getPrice( )
```

The prices of all varieties are obtained by cycling through the enumeration using a **for** loop. Because there is a copy of **price** for each enumeration constant, the value associated with one constant is separate and distinct from the value associated with another constant. This is a powerful concept, which is only available when enumerations are implemented as classes, as Java does.

Although the preceding example contains only one constructor, an **enum** can offer two or more overloaded forms, just as can any other class. For example, this version of **Apple** provides a default constructor that initializes the price to **-1**, to indicate that no price data is available:

```

// Use an enum constructor.
enum Apple {
    Jonathan(10), GoldenDel(9), RedDel, Winesap(15), Cortland(8);

    private int price; // price of each apple
}

```

```
// Constructor  
Apple(int p) { price = p; }  
  
// Overloaded constructor  
Apple() { price = -1; }  
  
int getPrice() { return price; }  
}
```

Notice that in this version, **RedDel** is not given an argument. This means that the default constructor is called, and **RedDel**'s price variable is given the value `-1`.

Here are two restrictions that apply to enumerations. First, an enumeration can't inherit another class. Second, an **enum** cannot be a superclass. This means that an **enum** can't be extended. Otherwise, **enum** acts much like any other class type. The key is to remember that each of the enumeration constants is an object of the class in which it is defined.

Enumerations Inherit **Enum**

Although you can't inherit a superclass when declaring an **enum**, all enumerations automatically inherit one: **java.lang.Enum**. This class defines several methods that are available for use by all enumerations. The **Enum** class is described in detail in Part II, but three of its methods warrant a discussion at this time.

You can obtain a value that indicates an enumeration constant's position in the list of constants. This is called its *ordinal value*, and it is retrieved by calling the **ordinal()** method, shown here:

```
final int ordinal()
```

It returns the ordinal value of the invoking constant. Ordinal values begin at zero. Thus, in the **Apple** enumeration, **Jonathan** has an ordinal value of zero, **GoldenDel** has an ordinal value of 1, **RedDel** has an ordinal value of 2, and so on.

You can compare the ordinal value of two constants of the same enumeration by using the **compareTo()** method. It has this general form:

```
final int compareTo(enum-type e)
```

Here, *enum-type* is the type of the enumeration, and *e* is the constant being compared to the invoking constant. Remember, both the invoking constant and *e* must be of the same enumeration. If the invoking constant has an ordinal value less than *e*'s, then **compareTo()** returns a negative value. If the two ordinal values are the same, then zero is returned. If the invoking constant has an ordinal value greater than *e*'s, then a positive value is returned.

You can compare for equality an enumeration constant with any other object by using **equals()**, which overrides the **equals()** method defined by **Object**. Although **equals()** can compare an enumeration constant to any other object, those two objects will be equal only if they both refer to the same constant, within the same enumeration. Simply having ordinal values in common will not cause **equals()** to return true if the two constants are from different enumerations.

Remember, you can compare two enumeration references for equality by using `= =`.

The following program demonstrates the `ordinal()`, `compareTo()`, and `equals()` methods:

```
// Demonstrate ordinal(), compareTo(), and equals().  
  
// An enumeration of apple varieties.  
enum Apple {  
    Jonathan, GoldenDel, RedDel, Winesap, Cortland  
}  
  
class EnumDemo4 {  
    public static void main(String args[])  
    {  
        Apple ap, ap2, ap3;  
  
        // Obtain all ordinal values using ordinal().  
        System.out.println("Here are all apple constants" +  
                           " and their ordinal values: ");  
        for(Apple a : Apple.values())  
            System.out.println(a + " " + a.ordinal());  
  
        ap = Apple.RedDel;  
        ap2 = Apple.GoldenDel;  
        ap3 = Apple.RedDel;  
  
        System.out.println();  
  
        // Demonstrate compareTo() and equals()  
        if(ap.compareTo(ap2) < 0)  
            System.out.println(ap + " comes before " + ap2);  
  
        if(ap.compareTo(ap2) > 0)  
            System.out.println(ap2 + " comes before " + ap);  
  
        if(ap.compareTo(ap3) == 0)  
            System.out.println(ap + " equals " + ap3);  
  
        System.out.println();  
  
        if(ap.equals(ap2))  
            System.out.println("Error!");  
  
        if(ap.equals(ap3))  
            System.out.println(ap + " equals " + ap3);  
  
        if(ap == ap3)  
            System.out.println(ap + " == " + ap3);  
    }  
}
```

The output from the program is shown here:

```
Here are all apple constants and their ordinal values:  
Jonathan 0
```

```
GoldenDel 1
RedDel 2
Winesap 3
Cortland 4

GoldenDel comes before RedDel
RedDel equals RedDel

RedDel equals RedDel
RedDel == RedDel
```

Another Enumeration Example

Before moving on, we will look at a different example that uses an **enum**. In Chapter 9, an automated “decision maker” program was created. In that version, variables called **NO**, **YES**, **MAYBE**, **LATER**, **SOON**, and **NEVER** were declared within an interface and used to represent the possible answers. While there is nothing technically wrong with that approach, the enumeration is a better choice. Here is an improved version of that program that uses an **enum** called **Answers** to define the answers. You should compare this version to the original in Chapter 9.

```
// An improved version of the "Decision Maker"
// program from Chapter 9. This version uses an
// enum, rather than interface variables, to
// represent the answers.

import java.util.Random;

// An enumeration of the possible answers.
enum Answers {
    NO, YES, MAYBE, LATER, SOON, NEVER
}

class Question {
    Random rand = new Random();
    Answers ask() {
        int prob = (int) (100 * rand.nextDouble());

        if (prob < 15)
            return Answers.MAYBE; // 15%
        else if (prob < 30)
            return Answers.NO; // 15%
        else if (prob < 60)
            return Answers.YES; // 30%
        else if (prob < 75)
            return Answers.LATER; // 15%
        else if (prob < 98)
            return Answers.SOON; // 13%
        else
            return Answers.NEVER; // 2%
    }
}
```

```

class AskMe {
    static void answer(Answers result) {
        switch(result) {
            case NO:
                System.out.println("No");
                break;
            case YES:
                System.out.println("Yes");
                break;
            case MAYBE:
                System.out.println("Maybe");
                break;
            case LATER:
                System.out.println("Later");
                break;
            case SOON:
                System.out.println("Soon");
                break;
            case NEVER:
                System.out.println("Never");
                break;
        }
    }

    public static void main(String args[]) {
        Question q = new Question();
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
    }
}

```

Type Wrappers

As you know, Java uses primitive types (also called simple types), such as `int` or `double`, to hold the basic data types supported by the language. Primitive types, rather than objects, are used for these quantities for the sake of performance. Using objects for these values would add an unacceptable overhead to even the simplest of calculations. Thus, the primitive types are not part of the object hierarchy, and they do not inherit `Object`.

Despite the performance benefit offered by the primitive types, there are times when you will need an object representation. For example, you can't pass a primitive type by reference to a method. Also, many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types. To handle these (and other) situations, Java provides *type wrappers*, which are classes that encapsulate a primitive type within an object. The type wrapper classes are described in detail in Part II, but they are introduced here because they relate directly to Java's autoboxing feature.

The type wrappers are **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy. Each is briefly examined next.

Character

Character is a wrapper around a **char**. The constructor for **Character** is

```
Character(char ch)
```

Here, *ch* specifies the character that will be wrapped by the **Character** object being created.

To obtain the **char** value contained in a **Character** object, call **charValue()**, shown here:

```
char charValue()
```

It returns the encapsulated character.

Boolean

Boolean is a wrapper around **boolean** values. It defines these constructors:

```
Boolean(boolean boolValue)
```

```
Boolean(String boolString)
```

In the first version, *boolValue* must be either **true** or **false**. In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.

To obtain a **boolean** value from a **Boolean** object, use **booleanValue()**, shown here:

```
boolean booleanValue()
```

It returns the **boolean** equivalent of the invoking object.

The Numeric Type Wrappers

By far, the most commonly used type wrappers are those that represent numeric values. These are **Byte**, **Short**, **Integer**, **Long**, **Float**, and **Double**. All of the numeric type wrappers inherit the abstract class **Number**. **Number** declares methods that return the value of an object in each of the different number formats. These methods are shown here:

```
byte byteValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
short shortValue()
```

For example, **doubleValue()** returns the value of an object as a **double**, **floatValue()** returns the value as a **float**, and so on. These methods are implemented by each of the numeric type wrappers.

All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value. For example, here are the constructors defined for **Integer**:

```
Integer(int num)
Integer(String str)
```

If *str* does not contain a valid numeric value, then a **NumberFormatException** is thrown.

All of the type wrappers override **toString()**. It returns the human-readable form of the value contained within the wrapper. This allows you to output the value by passing a type wrapper object to **println()**, for example, without having to convert it into its primitive type.

The following program demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value.

```
// Demonstrate a type wrapper.
class Wrap {
    public static void main(String args[]) {
        Integer iOb = new Integer(100);
        int i = iOb.intValue();
        System.out.println(i + " " + iOb); // displays 100 100
    }
}
```

This program wraps the integer value 100 inside an **Integer** object called **iOb**. The program then obtains this value by calling **intValue()** and stores the result in **i**.

The process of encapsulating a value within an object is called *boxing*. Thus, in the program, this line boxes the value 100 into an **Integer**:

```
Integer iOb = new Integer(100);
```

The process of extracting a value from a type wrapper is called *unboxing*. For example, the program unboxes the value in **iOb** with this statement:

```
int i = iOb.intValue();
```

The same general procedure used by the preceding program to box and unbox values has been employed since the original version of Java. However, since JDK 5, Java fundamentally improved on this through the addition of autoboxing, described next.

Autoboxing

Beginning with JDK 5, Java added two important features: *autoboxing* and *auto-unboxing*. Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object. Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. There is no need to call a method such as **intValue()** or **doubleValue()**.

The addition of autoboxing and auto-unboxing greatly streamlines the coding of several algorithms, removing the tedium of manually boxing and unboxing values. It also helps prevent errors. Moreover, it is very important to generics, which operate only on objects. Finally, autoboxing makes working with the Collections Framework (described in Part II) much easier.

With autoboxing, it is no longer necessary to manually construct an object in order to wrap a primitive type. You need only assign that value to a type-wrapper reference. Java automatically constructs the object for you. For example, here is the modern way to construct an **Integer** object that has the value 100:

```
Integer iOb = 100; // autobox an int
```

Notice that the object is not explicitly created through the use of `new`. Java handles this for you, automatically.

To unbox an object, simply assign that object reference to a primitive-type variable. For example, to unbox `iOb`, you can use this line:

```
int i = iOb; // auto-unbox
```

Java handles the details for you.

Here is the preceding program rewritten to use autoboxing/unboxing:

```
// Demonstrate autoboxing/unboxing.
class AutoBox {
    public static void main(String args[]) {
        Integer iOb = 100; // autobox an int
        int i = iOb; // auto-unbox
        System.out.println(i + " " + iOb); // displays 100 100
    }
}
```

Autoboxing and Methods

In addition to the simple case of assignments, autoboxing automatically occurs whenever a primitive type must be converted into an object; auto-unboxing takes place whenever an object must be converted into a primitive type. Thus, autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method. For example, consider this:

```
// Autoboxing/unboxing takes place with
// method parameters and return values.

class AutoBox2 {
    // Take an Integer parameter and return
    // an int value;
    static int m(Integer v) {
        return v; // auto-unbox to int
    }

    public static void main(String args[]) {
        // Pass an int to m() and assign the return value
        // to an Integer. Here, the argument 100 is autoboxed
        // into an Integer. The return value is also autoboxed
        // into an Integer.
        Integer iOb = m(100);

        System.out.println(iOb);
    }
}
```

This program displays the following result:

In the program, notice that `m()` specifies an `Integer` parameter and returns an `int` result. Inside `main()`, `m()` is passed the value 100. Because `m()` is expecting an `Integer`, this value is automatically boxed. Then, `m()` returns the `int` equivalent of its argument. This causes `v` to be auto-unboxed. Next, this `int` value is assigned to `iOb` in `main()`, which causes the `int` return value to be autoboxed.

Autoboxing/Unboxing Occurs in Expressions

In general, autoboxing and unboxing take place whenever a conversion into an object or from an object is required. This applies to expressions. Within an expression, a numeric object is automatically unboxed. The outcome of the expression is reboxed, if necessary. For example, consider the following program:

```
// Autoboxing/unboxing occurs inside expressions.

class AutoBox3 {
    public static void main(String args[]) {
        Integer iOb, iOb2;
        int i;

        iOb = 100;
        System.out.println("Original value of iOb: " + iOb);

        // The following automatically unboxes iOb,
        // performs the increment, and then reboxes
        // the result back into iOb.
        ++iOb;
        System.out.println("After ++iOb: " + iOb);

        // Here, iOb is unboxed, the expression is
        // evaluated, and the result is reboxed and
        // assigned to iOb2.
        iOb2 = iOb + (iOb / 3);
        System.out.println("iOb2 after expression: " + iOb2);

        // The same expression is evaluated, but the
        // result is not reboxed.
        i = iOb + (iOb / 3);
        System.out.println("i after expression: " + i);
    }
}
```

The output is shown here:

```
Original value of iOb: 100
After ++iOb: 101
iOb2 after expression: 134
i after expression: 134
```

In the program, pay special attention to this line:

```
++iOb;
```

This causes the value in **iOb** to be incremented. It works like this: **iOb** is unboxed, the value is incremented, and the result is reboxed.

Auto-unboxing also allows you to mix different types of numeric objects in an expression. Once the values are unboxed, the standard type promotions and conversions are applied. For example, the following program is perfectly valid:

```
class AutoBox4 {  
    public static void main(String args[]) {  
  
        Integer iOb = 100;  
        Double dOb = 98.6;  
  
        dOb = dOb + iOb;  
        System.out.println("dOb after expression: " + dOb);  
    }  
}
```

The output is shown here:

```
dOb after expression: 198.6
```

As you can see, both the **Double** object **dOb** and the **Integer** object **iOb** participated in the addition, and the result was reboxed and stored in **dOb**.

Because of auto-unboxing, you can use **Integer** numeric objects to control a **switch** statement. For example, consider this fragment:

```
Integer iOb = 2;  
  
switch(iOb) {  
    case 1: System.out.println("one");  
    break;  
    case 2: System.out.println("two");  
    break;  
    default: System.out.println("error");  
}
```

When the **switch** expression is evaluated, **iOb** is unboxed and its **int** value is obtained.

As the examples in the program show, because of autoboxing/unboxing, using numeric objects in an expression is both intuitive and easy. In the past, such code would have involved casts and calls to methods such as **intValue()**.

Autoboxing/Unboxing Boolean and Character Values

As described earlier, Java also supplies wrappers for **boolean** and **char**. These are **Boolean** and **Character**. Autoboxing/unboxing applies to these wrappers, too. For example, consider the following program:

```
// Autoboxing/unboxing a Boolean and Character.

class AutoBox5 {
    public static void main(String args[]) {
        // Autobox/unbox a boolean.
        Boolean b = true;

        // Below, b is auto-unboxed when used in
        // a conditional expression, such as an if.
        if(b) System.out.println("b is true");

        // Autobox/unbox a char.
        Character ch = 'x'; // box a char
        char ch2 = ch; // unbox a char

        System.out.println("ch2 is " + ch2);
    }
}
```

The output is shown here:

```
b is true
ch2 is x
```

The most important thing to notice about this program is the auto-unboxing of **b** inside the **if** conditional expression. As you should recall, the conditional expression that controls an **if** must evaluate to type **boolean**. Because of auto-unboxing, the **boolean** value contained within **b** is automatically unboxed when the conditional expression is evaluated. Thus, with the advent of autoboxing/unboxing, a **Boolean** object can be used to control an **if** statement.

Because of auto-unboxing, a **Boolean** object can now also be used to control any of Java's loop statements. When a **Boolean** is used as the conditional expression of a **while**, **for**, or **do/while**, it is automatically unboxed into its **boolean** equivalent. For example, this is now perfectly valid code:

```
Boolean b;
// ...
while(b) { // ... }
```

Autoboxing/Unboxing Helps Prevent Errors

In addition to the convenience that it offers, autoboxing/unboxing can also help prevent errors. For example, consider the following program:

```
// An error produced by manual unboxing.
class UnboxingError {
    public static void main(String args[]) {
```

```
Integer iOb = 1000; // autobox the value 1000  
  
int i = iOb.byteValue(); // manually unbox as byte !!!  
  
System.out.println(i); // does not display 1000 !  
}  
}
```

This program displays not the expected value of 1000, but -24! The reason is that the value inside **iOb** is manually unboxed by calling **byteValue()**, which causes the truncation of the value stored in **iOb**, which is 1,000. This results in the garbage value of -24 being assigned to **i**. Auto-unboxing prevents this type of error because the value in **iOb** will always auto-unbox into a value compatible with **int**.

In general, because autoboxing always creates the proper object, and auto-unboxing always produces the proper value, there is no way for the process to produce the wrong type of object or value. In the rare instances where you want a type different than that produced by the automated process, you can still manually box and unbox values. Of course, the benefits of autoboxing/unboxing are lost. In general, new code should employ autoboxing/unboxing. It is the way that modern Java code is written.

A Word of Warning

Because of autoboxing and auto-unboxing, some might be tempted to use objects such as **Integer** or **Double** exclusively, abandoning primitives altogether. For example, with autoboxing/unboxing it is possible to write code like this:

```
// A bad use of autoboxing/unboxing!  
Double a, b, c;  
  
a = 10.0;  
b = 4.0;  
  
c = Math.sqrt(a*a + b*b);  
  
System.out.println("Hypotenuse is " + c);
```

In this example, objects of type **Double** hold values that are used to calculate the hypotenuse of a right triangle. Although this code is technically correct and does, in fact, work properly, it is a very bad use of autoboxing/unboxing. It is far less efficient than the equivalent code written using the primitive type **double**. The reason is that each autobox and auto-unbox adds overhead that is not present if the primitive type is used.

In general, you should restrict your use of the type wrappers to only those cases in which an object representation of a primitive type is required. Autoboxing/unboxing was not added to Java as a “back door” way of eliminating the primitive types.

Annotations (Metadata)

Since JDK 5, Java has supported a feature that enables you to embed supplemental information into a source file. This information, called an *annotation*, does not change the actions of a program. Thus, an annotation leaves the semantics of a program unchanged.

However, this information can be used by various tools during both development and deployment. For example, an annotation might be processed by a source-code generator. The term *metadata* is also used to refer to this feature, but the term *annotation* is the most descriptive and more commonly used.

Annotation Basics

An annotation is created through a mechanism based on the **interface**. Let's begin with an example. Here is the declaration for an annotation called **MyAnno**:

```
// A simple annotation type.  
@interface MyAnno {  
    String str();  
    int val();  
}
```

First, notice the @ that precedes the keyword **interface**. This tells the compiler that an annotation type is being declared. Next, notice the two members **str()** and **val()**. All annotations consist solely of method declarations. However, you don't provide bodies for these methods. Instead, Java implements these methods. Moreover, the methods act much like fields, as you will see.

An annotation cannot include an **extends** clause. However, all annotation types automatically extend the **Annotation** interface. Thus, **Annotation** is a super-interface of all annotations. It is declared within the **java.lang.annotation** package. It overrides **hashCode()**, **equals()**, and **toString()**, which are defined by **Object**. It also specifies **annotationType()**, which returns a **Class** object that represents the invoking annotation.

Once you have declared an annotation, you can use it to annotate something. Prior to JDK 8, annotations could be used only on declarations, and that is where we will begin. (JDK 8 adds the ability to annotate type use, and this is described later in this chapter. However, the same basic techniques apply to both kinds of annotations.) Any type of declaration can have an annotation associated with it. For example, classes, methods, fields, parameters, and **enum** constants can be annotated. Even an annotation can be annotated. In all cases, the annotation precedes the rest of the declaration.

When you apply an annotation, you give values to its members. For example, here is an example of **MyAnno** being applied to a method declaration:

```
// Annotate a method.  
@MyAnno(str = "Annotation Example", val = 100)  
public static void myMeth() { // ... }
```

This annotation is linked with the method **myMeth()**. Look closely at the annotation syntax. The name of the annotation, preceded by an @, is followed by a parenthesized list of member initializations. To give a member a value, that member's name is assigned a value. Therefore, in the example, the string "Annotation Example" is assigned to the **str** member of **MyAnno**. Notice that no parentheses follow **str** in this assignment. When an annotation member is given a value, only its name is used. Thus, annotation members look like fields in this context.

Specifying a Retention Policy

Before exploring annotations further, it is necessary to discuss *annotation retention policies*. A retention policy determines at what point an annotation is discarded. Java defines three such policies, which are encapsulated within the `java.lang.annotation.RetentionPolicy` enumeration. They are **SOURCE**, **CLASS**, and **RUNTIME**.

An annotation with a retention policy of **SOURCE** is retained only in the source file and is discarded during compilation.

An annotation with a retention policy of **CLASS** is stored in the `.class` file during compilation. However, it is not available through the JVM during run time.

An annotation with a retention policy of **RUNTIME** is stored in the `.class` file during compilation and is available through the JVM during run time. Thus, **RUNTIME** retention offers the greatest annotation persistence.

NOTE An annotation on a local variable declaration is not retained in the `.class` file.

A retention policy is specified for an annotation by using one of Java's built-in annotations: `@Retention`. Its general form is shown here:

`@Retention(retention-policy)`

Here, *retention-policy* must be one of the previously discussed enumeration constants. If no retention policy is specified for an annotation, then the default policy of **CLASS** is used.

The following version of `MyAnno` uses `@Retention` to specify the **RUNTIME** retention policy. Thus, `MyAnno` will be available to the JVM during program execution.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}
```

Obtaining Annotations at Run Time by Use of Reflection

Although annotations are designed mostly for use by other development or deployment tools, if they specify a retention policy of **RUNTIME**, then they can be queried at run time by any Java program through the use of *reflection*. Reflection is the feature that enables information about a class to be obtained at run time. The reflection API is contained in the `java.lang.reflect` package. There are a number of ways to use reflection, and we won't examine them all here. We will, however, walk through a few examples that apply to annotations.

The first step to using reflection is to obtain a `Class` object that represents the class whose annotations you want to obtain. `Class` is one of Java's built-in classes and is defined in `java.lang`. It is described in detail in Part II. There are various ways to obtain a `Class` object. One of the easiest is to call `getClass()`, which is a method defined by `Object`. Its general form is shown here:

```
final Class<?> getClass()
```

It returns the `Class` object that represents the invoking object.

NOTE Notice the <?> that follows **Class** in the declaration of **getClass()** just shown. This is related to Java's generics feature. **getClass()** and several other reflection-related methods discussed in this chapter make use of generics. Generics are described in Chapter 14. However, an understanding of generics is not needed to grasp the fundamental principles of reflection.

After you have obtained a **Class** object, you can use its methods to obtain information about the various items declared by the class, including its annotations. If you want to obtain the annotations associated with a specific item declared within a class, you must first obtain an object that represents that item. For example, **Class** supplies (among others) the **getMethod()**, **getField()**, and **getConstructor()** methods, which obtain information about a method, field, and constructor, respectively. These methods return objects of type **Method**, **Field**, and **Constructor**.

To understand the process, let's work through an example that obtains the annotations associated with a method. To do this, you first obtain a **Class** object that represents the class, and then call **getMethod()** on that **Class** object, specifying the name of the method. **getMethod()** has this general form:

```
Method getMethod(String methName, Class<?> ... paramTypes)
```

The name of the method is passed in *methName*. If the method has arguments, then **Class** objects representing those types must also be specified by *paramTypes*. Notice that *paramTypes* is a varargs parameter. This means that you can specify as many parameter types as needed, including zero. **getMethod()** returns a **Method** object that represents the method. If the method can't be found, **NoSuchMethodException** is thrown.

From a **Class**, **Method**, **Field**, or **Constructor** object, you can obtain a specific annotation associated with that object by calling **getAnnotation()**. Its general form is shown here:

```
<A extends Annotation> getAnnotation(Class<A> annoType)
```

Here, *annoType* is a **Class** object that represents the annotation in which you are interested. The method returns a reference to the annotation. Using this reference, you can obtain the values associated with the annotation's members. The method returns **null** if the annotation is not found, which will be the case if the annotation does not have **RUNTIME** retention.

Here is a program that assembles all of the pieces shown earlier and uses reflection to display the annotation associated with a method:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// An annotation type declaration.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

class Meta {

    // Annotate a method.
    @MyAnno(str = "Annotation Example", val = 100)
```

```
public static void myMeth() {  
    Meta ob = new Meta();  
  
    // Obtain the annotation for this method  
    // and display the values of the members.  
    try {  
        // First, get a Class object that represents  
        // this class.  
        Class<?> c = ob.getClass();  
  
        // Now, get a Method object that represents  
        // this method.  
        Method m = c.getMethod("myMeth");  
  
        // Next, get the annotation for this class.  
        MyAnno anno = m.getAnnotation(MyAnno.class);  
  
        // Finally, display the values.  
        System.out.println(anno.str() + " " + anno.val());  
    } catch (NoSuchMethodException exc) {  
        System.out.println("Method Not Found.");  
    }  
}  
  
public static void main(String args[]) {  
    myMeth();  
}  
}
```

The output from the program is shown here:

```
Annotation Example 100
```

This program uses reflection as described to obtain and display the values of **str** and **val** in the **MyAnno** annotation associated with **myMeth()** in the **Meta** class. There are two things to pay special attention to. First, in this line

```
MyAnno anno = m.getAnnotation(MyAnno.class);
```

notice the expression **MyAnno.class**. This expression evaluates to a **Class** object of type **MyAnno**, the annotation. This construct is called a *class literal*. You can use this type of expression whenever a **Class** object of a known class is needed. For example, this statement could have been used to obtain the **Class** object for **Meta**:

```
Class<?> c = Meta.class;
```

Of course, this approach only works when you know the class name of an object in advance, which might not always be the case. In general, you can obtain a class literal for classes, interfaces, primitive types, and arrays. (Remember, the **<?>** syntax relates to Java's generics feature. It is described in Chapter 14.)

The second point of interest is the way the values associated with **str** and **val** are obtained when they are output by the following line:

```
System.out.println(anno.str() + " " + anno.val());
```

Notice that they are invoked using the method-call syntax. This same approach is used whenever the value of an annotation member is required.

A Second Reflection Example

In the preceding example, **myMeth()** has no parameters. Thus, when **getMethod()** was called, only the name **myMeth** was passed. However, to obtain a method that has parameters, you must specify class objects representing the types of those parameters as arguments to **getMethod()**. For example, here is a slightly different version of the preceding program:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

class Meta {

    // myMeth now has two arguments.
    @MyAnno(str = "Two Parameters", val = 19)
    public static void myMeth(String str, int i)
    {
        Meta ob = new Meta();

        try {
            Class<?> c = ob.getClass();

            // Here, the parameter types are specified.
            Method m = c.getMethod("myMeth", String.class, int.class);

            MyAnno anno = m.getAnnotation(MyAnno.class);

            System.out.println(anno.str() + " " + anno.val());
        } catch (NoSuchMethodException exc) {
            System.out.println("Method Not Found.");
        }
    }

    public static void main(String args[]) {
        myMeth("test", 10);
    }
}
```

The output from this version is shown here:

```
Two Parameters 19
```

In this version, `myMeth()` takes a `String` and an `int` parameter. To obtain information about this method, `getMethod()` must be called as shown here:

```
Method m = c.getMethod("myMeth", String.class, int.class);
```

Here, the `Class` objects representing `String` and `int` are passed as additional arguments.

Obtaining All Annotations

You can obtain all annotations that have **RUNTIME** retention that are associated with an item by calling `getAnnotations()` on that item. It has this general form:

```
Annotation[] getAnnotations()
```

It returns an array of the annotations. `getAnnotations()` can be called on objects of type **Class**, **Method**, **Constructor**, and **Field**, among others.

Here is another reflection example that shows how to obtain all annotations associated with a class and with a method. It declares two annotations. It then uses those annotations to annotate a class and a method.

```
// Show all annotations for a class and a method.
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

@Retention(RetentionPolicy.RUNTIME)
@interface What {
    String description();
}

@What(description = "An annotation test class")
@MyAnno(str = "Meta2", val = 99)
class Meta2 {

    @What(description = "An annotation test method")
    @MyAnno(str = "Testing", val = 100)
    public static void myMeth() {
        Meta2 ob = new Meta2();

        try {
            Annotation annos[] = ob.getClass().getAnnotations();

            // Display all annotations for Meta2.
            System.out.println("All annotations for Meta2:");
            for(Annotation a : annos)
                System.out.println(a);

            System.out.println();

            // Display all annotations for myMeth.
        }
    }
}
```

```

        Method m = ob.getClass( ).getMethod("myMeth");
        annos = m.getAnnotations();

        System.out.println("All annotations for myMeth:");
        for(Annotation a : annos)
            System.out.println(a);

    } catch (NoSuchMethodException exc) {
        System.out.println("Method Not Found.");
    }
}

public static void main(String args[]) {
    myMeth();
}
}

```

The output is shown here:

```

All annotations for Meta2:
@What(description=An annotation test class)
@MyAnno(str=Meta2, val=99)

All annotations for myMeth:
@What(description=An annotation test method)
@MyAnno(str=Testing, val=100)

```

The program uses `getAnnotations()` to obtain an array of all annotations associated with the **Meta2** class and with the `myMeth()` method. As explained, `getAnnotations()` returns an array of **Annotation** objects. Recall that **Annotation** is a super-interface of all annotation interfaces and that it overrides `toString()` in **Object**. Thus, when a reference to an **Annotation** is output, its `toString()` method is called to generate a string that describes the annotation, as the preceding output shows.

The AnnotatedElement Interface

The methods `getAnnotation()` and `getAnnotations()` used by the preceding examples are defined by the **AnnotatedElement** interface, which is defined in `java.lang.reflect`. This interface supports reflection for annotations and is implemented by the classes **Method**, **Field**, **Constructor**, **Class**, and **Package**, among others.

In addition to `getAnnotation()` and `getAnnotations()`, **AnnotatedElement** defines several other methods. Two have been available since JDK 5. The first is `getDeclaredAnnotations()`, which has this general form:

```
Annotation[ ] getDeclaredAnnotations()
```

It returns all non-inherited annotations present in the invoking object. The second is `isAnnotationPresent()`, which has this general form:

```
boolean isAnnotationPresent(Class<? extends Annotation> annoType)
```

It returns **true** if the annotation specified by `annoType` is associated with the invoking object. It returns **false** otherwise. To these, JDK 8 adds `getDeclaredAnnotation()`,

`getAnnotationsByType()`, and `getDeclaredAnnotationsByType()`. Of these, the last two automatically work with a repeated annotation.(Repeated annotations are discussed at the end of this chapter.)

Using Default Values

You can give annotation members default values that will be used if no value is specified when the annotation is applied. A default value is specified by adding a **default** clause to a member's declaration. It has this general form:

```
type member( ) default value;
```

Here, *value* must be of a type compatible with *type*.

Here is `@MyAnno` rewritten to include default values:

```
// An annotation type declaration that includes defaults.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str() default "Testing";
    int val() default 9000;
}
```

This declaration gives a default value of "Testing" to **str** and 9000 to **val**. This means that neither value needs to be specified when `@MyAnno` is used. However, either or both can be given values if desired. Therefore, following are the four ways that `@MyAnno` can be used:

```
@MyAnno() // both str and val default
@MyAnno(str = "some string") // val defaults
@MyAnno(val = 100) // str defaults
@MyAnno(str = "Testing", val = 100) // no defaults
```

The following program demonstrates the use of default values in an annotation.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// An annotation type declaration that includes defaults.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str() default "Testing";
    int val() default 9000;
}

class Meta3 {

    // Annotate a method using the default values.
    @MyAnno()
    public static void myMeth() {
        Meta3 ob = new Meta3();

        // Obtain the annotation for this method
        // and display the values of the members.
        try {
            Class<?> c = ob.getClass();
```

```

Method m = c.getMethod("myMeth");

MyAnno anno = m.getAnnotation(MyAnno.class);

System.out.println(anno.str() + " " + anno.val());
} catch (NoSuchMethodException exc) {
    System.out.println("Method Not Found.");
}
}

public static void main(String args[]) {
    myMeth();
}
}

```

The output is shown here:

```
Testing 9000
```

Marker Annotations

A *marker* annotation is a special kind of annotation that contains no members. Its sole purpose is to mark an item. Thus, its presence as an annotation is sufficient. The best way to determine if a marker annotation is present is to use the method **isAnnotationPresent()**, which is defined by the **AnnotatedElement** interface.

Here is an example that uses a marker annotation. Because a marker interface contains no members, simply determining whether it is present or absent is sufficient.

```

import java.lang.annotation.*;
import java.lang.reflect.*;

// A marker annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker { }

class Marker {

    // Annotate a method using a marker.
    // Notice that no ( ) is needed.
    @MyMarker
    public static void myMeth() {
        Marker ob = new Marker();

        try {
            Method m = ob.getClass().getMethod("myMeth");

            // Determine if the annotation is present.
            if(m.isAnnotationPresent(MyMarker.class))
                System.out.println("MyMarker is present.");

        } catch (NoSuchMethodException exc) {
            System.out.println("Method Not Found.");
        }
    }
}

```

```
public static void main(String args[]) {  
    myMeth();  
}  
}
```

The output, shown here, confirms that **@MyMarker** is present:

```
MyMarker is present.
```

In the program, notice that you do not need to follow **@MyMarker** with parentheses when it is applied. Thus, **@MyMarker** is applied simply by using its name, like this:

```
@MyMarker
```

It is not wrong to supply an empty set of parentheses, but they are not needed.

Single-Member Annotations

A *single-member* annotation contains only one member. It works like a normal annotation except that it allows a shorthand form of specifying the value of the member. When only one member is present, you can simply specify the value for that member when the annotation is applied—you don't need to specify the name of the member. However, in order to use this shorthand, the name of the member must be **value**.

Here is an example that creates and uses a single-member annotation:

```
import java.lang.annotation.*;  
import java.lang.reflect.*;  
  
// A single-member annotation.  
@Retention(RetentionPolicy.RUNTIME)  
@interface MySingle {  
    int value(); // this variable name must be value  
}  
  
class Single {  
  
    // Annotate a method using a single-member annotation.  
    @MySingle(100)  
    public static void myMeth() {  
        Single ob = new Single();  
  
        try {  
            Method m = ob.getClass().getMethod("myMeth");  
  
            MySingle anno = m.getAnnotation(MySingle.class);  
  
            System.out.println(anno.value()); // displays 100  
        } catch (NoSuchMethodException exc) {  
            System.out.println("Method Not Found.");  
        }  
    }  
}
```

```
public static void main(String args[]) {
    myMeth();
}
}
```

As expected, this program displays the value 100. In the program, `@MySingle` is used to annotate `myMeth()`, as shown here:

```
@MySingle(100)
```

Notice that `value` = need not be specified.

You can use the single-value syntax when applying an annotation that has other members, but those other members must all have default values. For example, here the value `xyz` is added, with a default value of zero:

```
@interface SomeAnno {
    int value();
    int xyz() default 0;
}
```

In cases in which you want to use the default for `xyz`, you can apply `@SomeAnno`, as shown next, by simply specifying the value of `value` by using the single-member syntax.

```
@SomeAnno(88)
```

In this case, `xyz` defaults to zero, and `value` gets the value 88. Of course, to specify a different value for `xyz` requires that both members be explicitly named, as shown here:

```
@SomeAnno(value = 88, xyz = 99)
```

Remember, whenever you are using a single-member annotation, the name of that member must be `value`.

The Built-In Annotations

Java defines many built-in annotations. Most are specialized, but nine are general purpose. Of these, four are imported from `java.lang.annotation`: `@Retention`, `@Documented`, `@Target`, and `@Inherited`. Five—`@Override`, `@Deprecated`, `@FunctionalInterface`, `@SafeVarargs`, and `@SuppressWarnings`—are included in `java.lang`. Each is described here.

NOTE To `java.lang.annotation`, JDK 8 adds the annotations `Repeatable` and `Native`. `Repeatable` supports repeatable annotations, as described later in this chapter. `Native` annotates a field that can be accessed by native code.

`@Retention`

`@Retention` is designed to be used only as an annotation to another annotation. It specifies the retention policy as described earlier in this chapter.

@Documented

The **@Documented** annotation is a marker interface that tells a tool that an annotation is to be documented. It is designed to be used only as an annotation to an annotation declaration.

@Target

The **@Target** annotation specifies the types of items to which an annotation can be applied. It is designed to be used only as an annotation to another annotation. **@Target** takes one argument, which is an array of constants of the **ElementType** enumeration. This argument specifies the types of declarations to which the annotation can be applied. The constants are shown here along with the type of declaration to which they correspond:

Target Constant	Annotation Can Be Applied To
ANNOTATION_TYPE	Another annotation
CONSTRUCTOR	Constructor
FIELD	Field
LOCAL_VARIABLE	Local variable
METHOD	Method
PACKAGE	Package
PARAMETER	Parameter
TYPE	Class, interface, or enumeration
TYPE_PARAMETER	Type parameter (Added by JDK 8.)
TYPE_USE	Type use (Added by JDK 8.)

You can specify one or more of these values in a **@Target** annotation. To specify multiple values, you must specify them within a braces-delimited list. For example, to specify that an annotation applies only to fields and local variables, you can use this **@Target** annotation:

```
@Target( { ElementType.FIELD, ElementType.LOCAL_VARIABLE } )
```

If you don't use **@Target**, then, except for type parameters, the annotation can be used on any declaration. For this reason, it is often a good idea to explicitly specify the target or targets so as to clearly indicate the intended uses of an annotation.

@Inherited

@Inherited is a marker annotation that can be used only on another annotation declaration. Furthermore, it affects only annotations that will be used on class declarations. **@Inherited** causes the annotation for a superclass to be inherited by a subclass. Therefore, when a request for a specific annotation is made to the subclass, if that annotation is not present in the subclass, then its superclass is checked. If that annotation is present in the superclass, and if it is annotated with **@Inherited**, then that annotation will be returned.

@Override

@Override is a marker annotation that can be used only on methods. A method annotated with **@Override** must override a method from a superclass. If it doesn't, a compile-time error will result. It is used to ensure that a superclass method is actually overridden, and not simply overloaded.

@Deprecated

@Deprecated is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form.

@FunctionalInterface

@FunctionalInterface is a marker annotation added by JDK 8 and designed for use on interfaces. It indicates that the annotated interface is a functional interface. A *functional interface* is an interface that contains one and only one abstract method. Functional interfaces are used by lambda expressions. (See Chapter 15 for details on functional interfaces and lambda expressions.) If the annotated interface is not a functional interface, a compilation error will be reported. It is important to understand that **@FunctionalInterface** is not needed to create a functional interface. Any interface with exactly one abstract method is, by definition, a functional interface. Thus, **@FunctionalInterface** is purely informational.

@SafeVarargs

@SafeVarargs is a marker annotation that can be applied to methods and constructors. It indicates that no unsafe actions related to a varargs parameter occur. It is used to suppress unchecked warnings on otherwise safe code as it relates to non-reifiable vararg types and parameterized array instantiation. (A non-reifiable type is, essentially, a generic type. Generics are described in Chapter 14.) It must be applied only to vararg methods or constructors that are **static** or **final**.

@SuppressWarnings

@SuppressWarnings specifies that one or more warnings that might be issued by the compiler are to be suppressed. The warnings to suppress are specified by name, in string form.

Type Annotations

Beginning with JDK 8, the places in which annotations can be used has been expanded. As mentioned earlier, annotations were originally allowed only on declarations. However, with the advent of JDK 8, annotations can also be specified in most cases in which a type is used. This expanded aspect of annotations is called *type annotation*. For example, you can annotate the return type of a method, the type of **this** within a method, a cast, array levels, an inherited class, and a **throws** clause. You can also annotate generic types, including generic type parameter bounds and generic type arguments. (See Chapter 14 for a discussion of generics.)

Type annotations are important because they enable tools to perform additional checks on code to help prevent errors. Understand that, as a general rule, **javac** will not perform these checks, itself. A separate tool is used for this purpose, although such a tool might operate as a compiler plug-in.

A type annotation must include **ElementType.TYPE_USE** as a target. (Recall that valid annotation targets are specified using the **@Target** annotation, as previously described.) A type annotation applies to the type that the annotation precedes. For example, assuming some type annotation called **@TypeAnno**, the following is legal:

```
void myMeth() throws @TypeAnno NullPointerException { // ...}
```

Here, **@TypeAnno** annotates **NullPointerException** in the **throws** clause.

You can also annotate the type of **this** (called the *receiver*). As you know, **this** is an implicit argument to all instance methods and it refers to the invoking object. To annotate its type requires the use of another new JDK 8 feature. Beginning with JDK 8, you can explicitly declare **this** as the first parameter to a method. In this declaration, the type of **this** must be the type of its class; for example:

```
class SomeClass {
    int myMeth(SomeClass this, int i, int j) { // ...}
```

Here, because **myMeth()** is a method defined by **SomeClass**, the type of **this** is **SomeClass**. Using this declaration, you can now annotate the type of **this**. For example, again assuming that **@TypeAnno** is a type annotation, the following is legal:

```
int myMeth(@TypeAnno SomeClass this, int i, int j) { // ...}
```

It is important to understand that it is not necessary to declare **this** unless you are annotating it. (If **this** is not declared, it is still implicitly passed. JDK 8 *does not* change this fact.) Also, explicitly declaring **this** does not change any aspect of the method's signature because **this** is implicitly declared, by default. Again, you will declare **this** only if you want to apply a type annotation to it. If you do declare **this**, it *must* be the first parameter.

The following program shows a number of the places that a type annotation can be used. It defines several annotations, of which several are for type annotation. The names and targets of the annotations are shown here:

Annotation	Target
@TypeAnno	ElementType.TYPE_USE
@MaxLen	ElementType.TYPE_USE
@NotZeroLen	ElementType.TYPE_USE
@Unique	ElementType.TYPE_USE
@What	ElementType.TYPE_PARAMETER
@EmptyOK	ElementType.FIELD
@Recommended	ElementType.METHOD

Notice that **@EmptyOK**, **@Recommended**, and **@What** are not type annotations. They are included for comparison purposes. Of special interest is **@What**, which is used to annotate a generic type parameter declaration and is another new annotation feature added by JDK 8. The comments in the program describe each use.

```
// Demonstrate several type annotations.
import java.lang.annotation.*;
import java.lang.reflect.*;

// A marker annotation that can be applied to a type.
@Target(ElementType.TYPE_USE)
@interface TypeAnno { }

// Another marker annotation that can be applied to a type.
@Target(ElementType.TYPE_USE)
@interface NotZeroLen {
}

// Still another marker annotation that can be applied to a type.
@Target(ElementType.TYPE_USE)
@interface Unique { }

// A parameterized annotation that can be applied to a type.
@Target(ElementType.TYPE_USE)
@interface MaxLen {
    int value();
}

// An annotation that can be applied to a type parameter.
@Target(ElementType.TYPE_PARAMETER)
@interface What {
    String description();
}

// An annotation that can be applied to a field declaration.
@Target(ElementType.FIELD)
@interface EmptyOK { }

// An annotation that can be applied to a method declaration.
@Target(ElementType.METHOD)
@interface Recommended { }

// Use an annotation on a type parameter.
class TypeAnnoDemo<@What(description = "Generic data type") T> {

    // Use a type annotation on a constructor.
    public @Unique TypeAnnoDemo() {}

    // Annotate the type (in this case String), not the field.
    @TypeAnno String str;
}
```

```
// This annotates the field test.  
@EmptyOK String test;  
  
// Use a type annotation to annotate this (the receiver).  
public int f(@TypeAnno TypeAnnoDemo<T> this, int x) {  
    return 10;  
}  
  
// Annotate the return type.  
public @TypeAnno Integer f2(int j, int k) {  
    return j+k;  
}  
  
// Annotate the method declaration.  
public @Recommended Integer f3(String str) {  
    return str.length() / 2;  
}  
  
// Use a type annotation with a throws clause.  
public void f4() throws @TypeAnno NullPointerException {  
    // ...  
}  
  
// Annotate array levels.  
String @MaxLen(10) [] @NotZeroLen [] w;  
  
// Annotate the array element type.  
@TypeAnno Integer[] vec;  
  
public static void myMeth(int i) {  
  
    // Use a type annotation on a type argument.  
    TypeAnnoDemo<@TypeAnno Integer> ob =  
        new TypeAnnoDemo<@TypeAnno Integer>();  
  
    // Use a type annotation with new.  
    @Unique TypeAnnoDemo<Integer> ob2 = new @Unique TypeAnnoDemo<Integer>();  
  
    Object x = new Integer(10);  
    Integer y;  
  
    // Use a type annotation on a cast.  
    y = (@TypeAnno Integer) x;  
}  
  
public static void main(String args[]) {  
    myMeth(10);  
}  
  
// Use type annotation with inheritance clause.  
class SomeClass extends @TypeAnno TypeAnnoDemo<Boolean> {}  
}
```

Although what most of the annotations in the preceding program refer to is clear, four uses require a bit of discussion. The first is the annotation of a method return type versus the annotation of a method declaration. In the program, pay special attention to these two method declarations:

```
// Annotate the return type.
public @TypeAnno Integer f2(int j, int k) {
    return j+k;
}

// Annotate the method declaration.
public @Recommended Integer f3(String str) {
    return str.length() / 2;
}
```

Notice that in both cases, an annotation precedes the method's return type (which is `Integer`). However, the two annotations annotate two different things. In the first case, the `@TypeAnno` annotation annotates `f2()`'s return type. This is because `@TypeAnno` has its target specified as `ElementType.TYPE_USE`, which means that it can be used to annotate type uses. In the second case, `@Recommended` annotates the method declaration, itself. This is because `@Recommended` has its target specified as `ElementType.METHOD`. As a result, `@Recommended` applies to the declaration, not the return type. Therefore, the target specification is used to eliminate what, at first glance, appears to be ambiguity between the annotation of a method declaration and the annotation of the method's return type.

One other thing about annotating a method return type: You cannot annotate a return type of `void`.

The second point of interest are the field annotations, shown here:

```
// Annotate the type (in this case String), not the field.
@TypeAnno String str;

// This annotates the field test.
@EmptyOK String test;
```

Here, `@TypeAnno` annotates the type `String`, but `@EmptyOK` annotates the field `test`. Even though both annotations precede the entire declaration, their targets are different, based on the target element type. If the annotation has the `ElementType.TYPE_USE` target, then the type is annotated. If it has `ElementType_FIELD` as a target, then the field is annotated. Thus, the situation is similar to that just described for methods, and no ambiguity exists. The same mechanism also disambiguates annotations on local variables.

Next, notice how `this` (the receiver) is annotated here:

```
public int f(@TypeAnno TypeAnnoDemo<T> this, int x) {
```

Here, `this` is specified as the first parameter and is of type `TypeAnnoDemo` (which is the class of which `f()` is a member). As explained, beginning with JDK 8, an instance method declaration can explicitly specify the `this` parameter for the sake of applying a type annotation.

Finally, look at how array levels are annotated by the following statement:

```
String @MaxLen(10) [] @NotZeroLen [] w;
```

In this declaration, **@MaxLen** annotates the type of the first level and **@NotZeroLen** annotates the type of the second level. In this declaration

```
@TypeAnno Integer [] vec;
```

the element type **Integer** is annotated.

Repeating Annotations

Another new JDK 8 annotation feature enables an annotation to be repeated on the same element. This is called *repeating annotations*. For an annotation to be repeatable, it must be annotated with the **@Repeatable** annotation, defined in **java.lang.annotation**. Its **value** field specifies the *container* type for the repeatable annotation. The container is specified as an annotation for which the **value** field is an array of the repeatable annotation type. Thus, to create a repeatable annotation, you must create a container annotation and then specify that annotation type as an argument to the **@Repeatable** annotation.

To access the repeated annotations using a method such as **getAnnotation()**, you will use the container annotation, not the repeatable annotation. The following program shows this approach. It converts the version of **MyAnno** shown previously into a repeatable annotation and demonstrates its use.

```
// Demonstrate a repeated annotation.

import java.lang.annotation.*;
import java.lang.reflect.*;

// Make MyAnno repeatable.
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(MyRepeatedAnnos.class)
@interface MyAnno {
    String str() default "Testing";
    int val() default 9000;
}

// This is the container annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MyRepeatedAnnos {
    MyAnno[] value();
}

class RepeatAnno {

    // Repeat MyAnno on myMeth().
    @MyAnno(str = "First annotation", val = -1)
    @MyAnno(str = "Second annotation", val = 100)
    public static void myMeth(String str, int i)
```

```

{
    RepeatAnno ob = new RepeatAnno();

    try {
        Class<?> c = ob.getClass();

        // Obtain the annotations for myMeth().
        Method m = c.getMethod("myMeth", String.class, int.class);

        // Display the repeated MyAnno annotations.
        Annotation anno = m.getAnnotation(MyRepeatedAnnos.class);
        System.out.println(anno);

    } catch (NoSuchMethodException exc) {
        System.out.println("Method Not Found.");
    }
}

public static void main(String args[]) {
    myMeth("test", 10);
}
}

```

The output is shown here:

```
@MyRepeatedAnnos(value=[@MyAnno(str=First annotation, val=-1),
@MyAnno(str=Second annotation, val=100)])
```

As explained, in order for **MyAnno** to be repeatable, it must be annotated with the **@Repeatable** annotation, which specifies its container annotation. The container annotation is called **MyRepeatedAnnos**. The program accesses the repeated annotations by calling **getAnnotation()**, passing in the class of the container annotation, not the repeatable annotation, itself. As the output shows, the repeated annotations are separated by a comma. They are not returned individually.

Another way to obtain the repeated annotations is to use one of the new methods added to **AnnotatedElement** by JDK 8, which can operate directly on a repeated annotation. These are **getAnnotationsByType()** and **getDeclaredAnnotationsByType()**. Here, we will use the former. It is shown here:

```
<T extends Annotation> T[ ] getAnnotationsByType(Class<T> annoType)
```

It returns an array of the annotations of *annoType* associated with the invoking object. If no annotations are present, the array will be of zero length. Here is an example. Assuming the preceding program, the following sequence uses **getAnnotationsByType()** to obtain the repeated **MyAnno** annotations:

```
Annotation[] annos = m.getAnnotationsByType(MyAnno.class);
for(Annotation a : annos)
    System.out.println(a);
```

Here, the repeated annotation type, which is **MyAnno**, is passed to `getAnnotationsByType()`. The returned array contains all of the instances of **MyAnno** associated with `myMeth()`, which, in this example, is two. Each repeated annotation can be accessed via its index in the array. In this case, each **MyAnno** annotation is displayed via a for-each loop.

Some Restrictions

There are a number of restrictions that apply to annotation declarations. First, no annotation can inherit another. Second, all methods declared by an annotation must be without parameters. Furthermore, they must return one of the following:

- A primitive type, such as **int** or **double**
- An object of type **String** or **Class**
- An **enum** type
- Another annotation type
- An array of one of the preceding types

Annotations cannot be generic. In other words, they cannot take type parameters. (Generics are described in Chapter 14.) Finally, annotation methods cannot specify a **throws** clause.

This page has been intentionally left blank

CHAPTER

13

I/O, Applets, and Other Topics

This chapter introduces two of Java’s most important packages: `io` and `applet`. The `io` package supports Java’s basic I/O (input/output) system, including file I/O. The `applet` package supports applets. Support for both I/O and applets comes from Java’s core API libraries, not from language keywords. For this reason, an in-depth discussion of these topics is found in Part II of this book, which examines Java’s API classes. This chapter discusses the foundation of these two subsystems so that you can see how they are integrated into the Java language and how they fit into the larger context of the Java programming and execution environment. This chapter also examines the `try-with-resources` statement and the last of Java’s keywords: `transient`, `volatile`, `instanceof`, `native`, `strictfp`, and `assert`. It concludes by discussing static import, describing another use for the `this` keyword, and introducing compact profiles (a feature added by JDK 8).

I/O Basics

As you may have noticed while reading the preceding 12 chapters, not much use has been made of I/O in the example programs. In fact, aside from `print()` and `println()`, none of the I/O methods have been used significantly. The reason is simple: most real applications of Java are not text-based, console programs. Rather, they are either graphically oriented programs that rely on one of Java’s graphical user interface (GUI) frameworks, such as Swing, the AWT, or JavaFX, for user interaction, or they are Web applications. Although text-based, console programs are excellent as teaching examples, they do not constitute an important use for Java in the real world. Also, Java’s support for console I/O is limited and somewhat awkward to use—even in simple example programs. Text-based console I/O is just not that useful in real-world Java programming.

The preceding paragraph notwithstanding, Java does provide strong, flexible support for I/O as it relates to files and networks. Java’s I/O system is cohesive and consistent. In fact, once you understand its fundamentals, the rest of the I/O system is easy to master. A general overview of I/O is presented here. A detailed description is found in Chapters 20 and 21.

Streams

Java programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to different types of devices. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the `java.io` package.

NOTE In addition to the stream-based I/O defined in `java.io`, Java also provides buffer- and channel-based I/O, which is defined in `java.nio` and its subpackages. They are described in Chapter 21.

Byte Streams and Character Streams

Java defines two types of streams: byte and character. *Byte streams* provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. *Character streams* provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

The original version of Java (Java 1.0) did not include character streams and, thus, all I/O was byte-oriented. Character streams were added by Java 1.1, and certain byte-oriented classes and methods were deprecated. Although old code that doesn't use character streams is becoming increasingly rare, it may still be encountered from time to time. As a general rule, old code should be updated to take advantage of character streams where appropriate.

One other point: at the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters.

An overview of both byte-oriented streams and character-oriented streams is presented in the following sections.

The Byte Stream Classes

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**. Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connections, and even memory buffers. The byte stream classes in `java.io` are shown in Table 13-1. A few of these classes are discussed later in this section. Others are described in Part II of this book. Remember, to use the stream classes, you must import `java.io`.

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements InputStream
FilterOutputStream	Implements OutputStream
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains print() and println()
PushbackInputStream	Input stream that supports one-byte “unget,” which returns a byte to the input stream
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Table 13-1 The Byte Stream Classes in **java.io**

The abstract classes **InputStream** and **OutputStream** define several key methods that the other stream classes implement. Two of the most important are **read()** and **write()**, which, respectively, read and write bytes of data. Each has a form that is abstract and must be overridden by derived stream classes.

The Character Stream Classes

Character streams are defined by using two class hierarchies. At the top are two abstract classes: **Reader** and **Writer**. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these. The character stream classes in **java.io** are shown in Table 13-2.

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains print() and println()
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

Table 13-2 The Character Stream I/O Classes in **java.io**

The abstract classes **Reader** and **Writer** define several key methods that the other stream classes implement. Two of the most important methods are **read()** and **write()**, which read and write characters of data, respectively. Each has a form that is abstract and must be overridden by derived stream classes.

The Predefined Streams

As you know, all Java programs automatically import the **java.lang** package. This package defines a class called **System**, which encapsulates several aspects of the run-time environment. For example, using some of its methods, you can obtain the current time and the settings of various properties associated with the system. **System** also contains three predefined stream variables: **in**, **out**, and **err**. These fields are declared as **public**, **static**, and **final** within **System**. This means that they can be used by any other part of your program and without reference to a specific **System** object.

System.out refers to the standard output stream. By default, this is the console. **System.in** refers to standard input, which is the keyboard by default. **System.err** refers to the standard error stream, which also is the console by default. However, these streams may be redirected to any compatible I/O device.

`System.in` is an object of type **InputStream**; `System.out` and `System.err` are objects of type **PrintStream**. These are byte streams, even though they are typically used to read and write characters from and to the console. As you will see, you can wrap these within character-based streams, if desired.

The preceding chapters have been using `System.out` in their examples. You can use `System.err` in much the same way. As explained in the next section, use of `System.in` is a little more complicated.

Reading Console Input

In Java 1.0, the only way to perform console input was to use a byte stream. Today, using a byte stream to read console input is still acceptable. However, for commercial applications, the preferred method of reading console input is to use a character-oriented stream. This makes your program easier to internationalize and maintain.

In Java, console input is accomplished by reading from `System.in`. To obtain a character-based stream that is attached to the console, wrap `System.in` in a **BufferedReader** object. **BufferedReader** supports a buffered input stream. A commonly used constructor is shown here:

```
BufferedReader(Reader inputReader)
```

Here, `inputReader` is the stream that is linked to the instance of **BufferedReader** that is being created. **Reader** is an abstract class. One of its concrete subclasses is **InputStreamReader**, which converts bytes to characters. To obtain an **InputStreamReader** object that is linked to `System.in`, use the following constructor:

```
InputStreamReader(InputStream inputStream)
```

Because `System.in` refers to an object of type **InputStream**, it can be used for `inputStream`. Putting it all together, the following line of code creates a **BufferedReader** that is connected to the keyboard:

```
BufferedReader br = new BufferedReader(new  
    InputStreamReader(System.in));
```

After this statement executes, `br` is a character-based stream that is linked to the console through `System.in`.

Reading Characters

To read a character from a **BufferedReader**, use `read()`. The version of `read()` that we will be using is

```
int read() throws IOException
```

Each time that `read()` is called, it reads a character from the input stream and returns it as an integer value. It returns `-1` when the end of the stream is encountered. As you can see, it can throw an **IOException**.

The following program demonstrates `read()` by reading characters from the console until the user types a "q." Notice that any I/O exceptions that might be generated are simply thrown out of `main()`. Such an approach is common when reading from the console

in simple example programs such as those shown in this book, but in more sophisticated applications, you can handle the exceptions explicitly.

```
// Use a BufferedReader to read characters from the console.
import java.io.*;

class BRRead {
    public static void main(String args[]) throws IOException
    {
        char c;
        BufferedReader br = new
            BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter characters, 'q' to quit.");
        // read characters
        do {
            c = (char) br.read();
            System.out.println(c);
        } while(c != 'q');
    }
}
```

Here is a sample run:

```
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
```

This output may look a little different from what you expected because **System.in** is line buffered, by default. This means that no input is actually passed to the program until you press ENTER. As you can guess, this does not make **read()** particularly valuable for interactive console input.

Reading Strings

To read a string from the keyboard, use the version of **readLine()** that is a member of the **BufferedReader** class. Its general form is shown here:

```
String readLine() throws IOException
```

As you can see, it returns a **String** object.

The following program demonstrates **BufferedReader** and the **readLine()** method; the program reads and displays lines of text until you enter the word "stop":

```
// Read a string from console using a BufferedReader.
import java.io.*;
```

```

class BRReadLines {
    public static void main(String args[]) throws IOException
    {
        // create a BufferedReader using System.in
        BufferedReader br = new BufferedReader(new
                                                InputStreamReader(System.in));
        String str;
        System.out.println("Enter lines of text.");
        System.out.println("Enter 'stop' to quit.");
        do {
            str = br.readLine();
            System.out.println(str);
        } while(!str.equals("stop"));
    }
}

```

The next example creates a tiny text editor. It creates an array of **String** objects and then reads in lines of text, storing each line in the array. It will read up to 100 lines or until you enter "stop." It uses a **BufferedReader** to read from the console.

```

// A tiny editor.
import java.io.*;

class TinyEdit {
    public static void main(String args[]) throws IOException
    {
        // create a BufferedReader using System.in
        BufferedReader br = new BufferedReader(new
                                                InputStreamReader(System.in));
        String str[] = new String[100];
        System.out.println("Enter lines of text.");
        System.out.println("Enter 'stop' to quit.");
        for(int i=0; i<100; i++) {
            str[i] = br.readLine();
            if(str[i].equals("stop")) break;
        }
        System.out.println("\nHere is your file:");
        // display the lines
        for(int i=0; i<100; i++) {
            if(str[i].equals("stop")) break;
            System.out.println(str[i]);
        }
    }
}

```

Here is a sample run:

```

Enter lines of text.
Enter 'stop' to quit.
This is line one.
This is line two.
Java makes working with strings easy.
Just create String objects.

```

```

stop
Here is your file:
This is line one.
This is line two.
Java makes working with strings easy.
Just create String objects.

```

Writing Console Output

Console output is most easily accomplished with `print()` and `println()`, described earlier, which are used in most of the examples in this book. These methods are defined by the class `PrintStream` (which is the type of object referenced by `System.out`). Even though `System.out` is a byte stream, using it for simple program output is still acceptable. However, a character-based alternative is described in the next section.

Because `PrintStream` is an output stream derived from `OutputStream`, it also implements the low-level method `write()`. Thus, `write()` can be used to write to the console. The simplest form of `write()` defined by `PrintStream` is shown here:

```
void write(int byteval)
```

This method writes the byte specified by `byteval`. Although `byteval` is declared as an integer, only the low-order eight bits are written. Here is a short example that uses `write()` to output the character "A" followed by a newline to the screen:

```

// Demonstrate System.out.write().
class WriteDemo {
    public static void main(String args[]) {
        int b;

        b = 'A';
        System.out.write(b);
        System.out.write('\n');
    }
}

```

You will not often use `write()` to perform console output (although doing so might be useful in some situations) because `print()` and `println()` are substantially easier to use.

The PrintWriter Class

Although using `System.out` to write to the console is acceptable, its use is probably best for debugging purposes or for sample programs, such as those found in this book. For real-world programs, the recommended method of writing to the console when using Java is through a `PrintWriter` stream. `PrintWriter` is one of the character-based classes. Using a character-based class for console output makes internationalizing your program easier.

`PrintWriter` defines several constructors. The one we will use is shown here:

```
PrintWriter(OutputStream outputStream, boolean flushingOn)
```

Here, `outputStream` is an object of type `OutputStream`, and `flushingOn` controls whether Java flushes the output stream every time a `println()` method (among others) is called. If `flushingOn` is `true`, flushing automatically takes place. If `false`, flushing is not automatic.

PrintWriter supports the **print()** and **println()** methods. Thus, you can use these methods in the same way as you used them with **System.out**. If an argument is not a simple type, the **PrintWriter** methods call the object's **toString()** method and then display the result.

To write to the console by using a **PrintWriter**, specify **System.out** for the output stream and automatic flushing. For example, this line of code creates a **PrintWriter** that is connected to console output:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

The following application illustrates using a **PrintWriter** to handle console output:

```
// Demonstrate PrintWriter
import java.io.*;

public class PrintWriterDemo {
    public static void main(String args[]) {
        PrintWriter pw = new PrintWriter(System.out, true);

        pw.println("This is a string");
        int i = -7;
        pw.println(i);
        double d = 4.5e-7;
        pw.println(d);
    }
}
```

The output from this program is shown here:

```
This is a string
-7
4.5E-7
```

Remember, there is nothing wrong with using **System.out** to write simple text output to the console when you are learning Java or debugging your programs. However, using a **PrintWriter** makes your real-world applications easier to internationalize. Because no advantage is gained by using a **PrintWriter** in the sample programs shown in this book, we will continue to use **System.out** to write to the console.

Reading and Writing Files

Java provides a number of classes and methods that allow you to read and write files. Before we begin, it is important to state that the topic of file I/O is quite large and file I/O is examined in detail in Part II. The purpose of this section is to introduce the basic techniques that read from and write to a file. Although bytes streams are used, these techniques can be adapted to the character-based streams.

Two of the most often-used stream classes are **FileInputStream** and **FileOutputStream**, which create byte streams linked to files. To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor. Although both classes support additional constructors, the following are the forms that we will be using:

```
FileInputStream(String fileName) throws FileNotFoundException
FileOutputStream(String fileName) throws FileNotFoundException
```

Here, *fileName* specifies the name of the file that you want to open. When you create an input stream, if the file does not exist, then **FileNotFoundException** is thrown. For output streams, if the file cannot be opened or created, then **FileNotFoundException** is thrown. **FileNotFoundException** is a subclass of **IOException**. When an output file is opened, any preexisting file by the same name is destroyed.

NOTE In situations in which a security manager is present, several of the file classes, including **FileInputStream** and **FileOutputStream**, will throw a **SecurityException** if a security violation occurs when attempting to open a file. By default, applications run via **Java** do not use a security manager. For that reason, the I/O examples in this book do not need to watch for a possible **SecurityException**. However, other types of applications (such as applets) will use the security manager, and file I/O performed by such an application could generate a **SecurityException**. In that case, you will need to appropriately handle this exception.

When you are done with a file, you must close it. This is done by calling the **close()** method, which is implemented by both **FileInputStream** and **FileOutputStream**. It is shown here:

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

Closing a file releases the system resources allocated to the file, allowing them to be used by another file. Failure to close a file can result in “memory leaks” because of unused resources remaining allocated.

NOTE Beginning with JDK 7, the **close()** method is specified by the **AutoCloseable** interface in **java.lang.AutoCloseable** is inherited by the **Closeable** interface in **java.io**. Both interfaces are implemented by the stream classes, including **FileInputStream** and **FileOutputStream**.

Before moving on, it is important to point out that there are two basic approaches that you can use to close a file when you are done with it. The first is the traditional approach, in which **close()** is called explicitly when the file is no longer needed. This is the approach used by all versions of Java prior to JDK 7 and is, therefore, found in all pre-JDK 7 legacy code. The second is to use the **try-with-resources** statement added by JDK 7, which automatically closes a file when it is no longer needed. In this approach, no explicit call to **close()** is executed. Since there is a large amount of pre-JDK 7 legacy code that is still being used and maintained, it is important that you know and understand the traditional approach. Therefore, we will begin with it. The new automated approach is described in the following section.

To read from a file, you can use a version of **read()** that is defined within **FileInputStream**. The one that we will use is shown here:

```
int read() throws IOException
```

Each time that it is called, it reads a single byte from the file and returns the byte as an integer value. **read()** returns **-1** when the end of the file is encountered. It can throw an **IOException**.

The following program uses **read()** to input and display the contents of a file that contains ASCII text. The name of the file is specified as a command-line argument.

```
/* Display a text file.  
To use this program, specify the name  
of the file that you want to see.  
For example, to see a file called TEST.TXT,  
use the following command line.  
  
        java ShowFile TEST.TXT  
*/  
  
import java.io.*;  
  
class ShowFile {  
    public static void main(String args[])  
    {  
        int i;  
        FileInputStream fin;  
  
        // First, confirm that a filename has been specified.  
        if(args.length != 1) {  
            System.out.println("Usage: ShowFile filename");  
            return;  
        }  
  
        // Attempt to open the file.  
        try {  
            fin = new FileInputStream(args[0]);  
        } catch(FileNotFoundException e) {  
            System.out.println("Cannot Open File");  
            return;  
        }  
  
        // At this point, the file is open and can be read.  
        // The following reads characters until EOF is encountered.  
        try {  
            do {  
                i = fin.read();  
                if(i != -1) System.out.print((char) i);  
            } while(i != -1);  
        } catch(IOException e) {  
            System.out.println("Error Reading File");  
        }  
  
        // Close the file.  
        try {  
            fin.close();  
        } catch(IOException e) {  
            System.out.println("Error Closing File");  
        }  
    }  
}
```

In the program, notice the **try/catch** blocks that handle the I/O errors that might occur. Each I/O operation is monitored for exceptions, and if an exception occurs, it is handled. Be aware that in simple programs or example code, it is common to see I/O exceptions simply thrown out of **main()**, as was done in the earlier console I/O examples. Also, in some real-world code, it can be helpful to let an exception propagate to a calling routine to let the caller know that an I/O operation failed. However, most of the file I/O examples in this book handle all I/O exceptions explicitly, as shown, for the sake of illustration.

Although the preceding example closes the file stream after the file is read, there is a variation that is often useful. The variation is to call **close()** within a **finally** block. In this approach, all of the methods that access the file are contained within a **try** block, and the **finally** block is used to close the file. This way, no matter how the **try** block terminates, the file is closed. Assuming the preceding example, here is how the **try** block that reads the file can be recoded:

```
try {
    do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
    } while(i != -1);
} catch(IOException e) {
    System.out.println("Error Reading File");
} finally {
    // Close file on the way out of the try block.
    try {
        fin.close();
    } catch(IOException e) {
        System.out.println("Error Closing File");
    }
}
```

Although not an issue in this case, one advantage to this approach in general is that if the code that accesses a file terminates because of some non-I/O related exception, the file is still closed by the **finally** block.

Sometimes it's easier to wrap the portions of a program that open the file and access the file within a single **try** block (rather than separating the two) and then use a **finally** block to close the file. For example, here is another way to write the **ShowFile** program:

```
/* Display a text file.
To use this program, specify the name
of the file that you want to see.
For example, to see a file called TEST.TXT,
use the following command line.

java ShowFile TEST.TXT

This variation wraps the code that opens and
accesses the file within a single try block.
The file is closed by the finally block.
*/
```

```

import java.io.*;

class ShowFile {
    public static void main(String args[])
    {
        int i;
        FileInputStream fin = null;

        // First, confirm that a filename has been specified.
        if(args.length != 1) {
            System.out.println("Usage: ShowFile filename");
            return;
        }

        // The following code opens a file, reads characters until EOF
        // is encountered, and then closes the file via a finally block.
        try {
            fin = new FileInputStream(args[0]);

            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);

        } catch(FileNotFoundException e) {
            System.out.println("File Not Found.");
        } catch(IOException e) {
            System.out.println("An I/O Error Occurred");
        } finally {
            // Close file in all cases.
            try {
                if(fin != null) fin.close();
            } catch(IOException e) {
                System.out.println("Error Closing File");
            }
        }
    }
}

```

In this approach, notice that **fin** is initialized to **null**. Then, in the **finally** block, the file is closed only if **fin** is not **null**. This works because **fin** will be non-**null** only if the file is successfully opened. Thus, **close()** is not called if an exception occurs while opening the file.

It is possible to make the **try/catch** sequence in the preceding example a bit more compact. Because **FileNotFoundException** is a subclass of **IOException**, it need not be caught separately. For example, here is the sequence recoded to eliminate catching **FileNotFoundException**. In this case, the standard exception message, which describes the error, is displayed.

```

try {
    fin = new FileInputStream(args[0]);
    do {

```

```

        i = fin.read();
        if(i != -1) System.out.print((char) i);
    } while(i != -1);

} catch(IOException e) {
    System.out.println("I/O Error: " + e);
} finally {
    // Close file in all cases.
    try {
        if(fin != null) fin.close();
    } catch(IOException e) {
        System.out.println("Error Closing File");
    }
}
}

```

In this approach, any error, including an error opening the file, is simply handled by the single **catch** statement. Because of its compactness, this approach is used by many of the I/O examples in this book. Be aware, however, that this approach is not appropriate in cases in which you want to deal separately with a failure to open a file, such as might be caused if a user mistypes a filename. In such a situation, you might want to prompt for the correct name, for example, before entering a **try** block that accesses the file.

To write to a file, you can use the **write()** method defined by **FileOutputStream**. Its simplest form is shown here:

```
void write(int byteval) throws IOException
```

This method writes the byte specified by *byteval* to the file. Although *byteval* is declared as an integer, only the low-order eight bits are written to the file. If an error occurs during writing, an **IOException** is thrown. The next example uses **write()** to copy a file:

```

/*
 * Copy a file.
 * To use this program, specify the name
 * of the source file and the destination file.
 * For example, to copy a file called FIRST.TXT
 * to a file called SECOND.TXT, use the following
 * command line.

java CopyFile FIRST.TXT SECOND.TXT
*/
import java.io.*;

class CopyFile {
    public static void main(String args[]) throws IOException
    {
        int i;
        FileInputStream fin = null;
        FileOutputStream fout = null;

        // First, confirm that both files have been specified.
        if(args.length != 2) {
            System.out.println("Usage: CopyFile from to");
            return;
        }
        fin = new FileInputStream(args[0]);
        fout = new FileOutputStream(args[1]);
        while((i = fin.read()) != -1)
            fout.write(i);
        fin.close();
        fout.close();
    }
}

```

```
}

// Copy a File.
try {
    // Attempt to open the files.
    fin = new FileInputStream(args[0]);
    fout = new FileOutputStream(args[1]);

    do {
        i = fin.read();
        if(i != -1) fout.write(i);
    } while(i != -1);

} catch(IOException e) {
    System.out.println("I/O Error: " + e);
} finally {
    try {
        if(fin != null) fin.close();
    } catch(IOException e2) {
        System.out.println("Error Closing Input File");
    }
    try {
        if(fout != null) fout.close();
    } catch(IOException e2) {
        System.out.println("Error Closing Output File");
    }
}
}
```

In the program, notice that two separate **try** blocks are used when closing the files. This ensures that both files are closed, even if the call to **fin.close()** throws an exception.

In general, notice that all potential I/O errors are handled in the preceding two programs by the use of exceptions. This differs from some computer languages that use error codes to report file errors. Not only do exceptions make file handling cleaner, but they also enable Java to easily differentiate the end-of-file condition from file errors when input is being performed.

Automatically Closing a File

In the preceding section, the example programs have made explicit calls to **close()** to close a file once it is no longer needed. As mentioned, this is the way files were closed when using versions of Java prior to JDK 7. Although this approach is still valid and useful, JDK 7 added a new feature that offers another way to manage resources, such as file streams, by automating the closing process. This feature, sometimes referred to as *automatic resource management*, or *ARM* for short, is based on an expanded version of the **try** statement. The principal advantage of automatic resource management is that it prevents situations in which a file (or other resource) is inadvertently not released after it is no longer needed. As explained, forgetting to close a file can result in memory leaks, and could lead to other problems.

Automatic resource management is based on an expanded form of the **try** statement. Here is its general form:

```
try (resource-specification) {
    // use the resource
}
```

Here, *resource-specification* is a statement that declares and initializes a resource, such as a file stream. It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the **try** block ends, the resource is automatically released. In the case of a file, this means that the file is automatically closed. (Thus, there is no need to call **close()** explicitly.) Of course, this form of **try** can also include **catch** and **finally** clauses. This new form of **try** is called the *try-with-resources* statement.

The **try-with-resources** statement can be used only with those resources that implement the **AutoCloseable** interface defined by **java.lang**. This interface defines the **close()** method. **AutoCloseable** is inherited by the **Closeable** interface in **java.io**. Both interfaces are implemented by the stream classes. Thus, **try-with-resources** can be used when working with streams, including file streams.

As a first example of automatically closing a file, here is a reworked version of the **ShowFile** program that uses it:

```
/* This version of the ShowFile program uses a try-with-resources
   statement to automatically close a file after it is no longer needed.

   Note: This code requires JDK 7 or later.
*/

import java.io.*;

class ShowFile {
    public static void main(String args[])
    {
        int i;

        // First, confirm that a filename has been specified.
        if(args.length != 1) {
            System.out.println("Usage: ShowFile filename");
            return;
        }

        // The following code uses a try-with-resources statement to open
        // a file and then automatically close it when the try block is left.
        try(FileInputStream fin = new FileInputStream(args[0])) {

            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);

        } catch(FileNotFoundException e) {
            System.out.println("File Not Found.");
        } catch(IOException e) {
```

```
        System.out.println("An I/O Error Occurred");
    }
}
```

In the program, pay special attention to how the file is opened within the `try` statement:

```
try(FileInputStream fin = new FileInputStream(args[0])) {
```

Notice how the resource-specification portion of the `try` declares a `FileInputStream` called `fin`, which is then assigned a reference to the file opened by its constructor. Thus, in this version of the program, the variable `fin` is local to the `try` block, being created when the `try` is entered. When the `try` is left, the stream associated with `fin` is automatically closed by an implicit call to `close()`. You don't need to call `close()` explicitly, which means that you can't forget to close the file. This is a key advantage of using `try-with-resources`.

It is important to understand that the resource declared in the `try` statement is implicitly **final**. This means that you can't assign to the resource after it has been created. Also, the scope of the resource is limited to the `try-with-resources` statement.

You can manage more than one resource within a single `try` statement. To do so, simply separate each resource specification with a semicolon. The following program shows an example. It reworks the `CopyFile` program shown earlier so that it uses a single `try-with-resources` statement to manage both `fin` and `fout`.

```
/* A version of CopyFile that uses try-with-resources.
   It demonstrates two resources (in this case files) being
   managed by a single try statement.
*/
import java.io.*;

class CopyFile {
    public static void main(String args[]) throws IOException
    {
        int i;

        // First, confirm that both files have been specified.
        if(args.length != 2) {
            System.out.println("Usage: CopyFile from to");
            return;
        }

        // Open and manage two files via the try statement.
        try (FileInputStream fin = new FileInputStream(args[0]);
             FileOutputStream fout = new FileOutputStream(args[1])) {
            do {
                i = fin.read();
                if(i != -1) fout.write(i);
            } while(i != -1);
        }
    }
}
```

```
    } catch(IOException e) {
        System.out.println("I/O Error: " + e);
    }
}
```

In this program, notice how the input and output files are opened within the **try** block:

```
try (FileInputStream fin = new FileInputStream(args[0]);
     FileOutputStream fout = new FileOutputStream(args[1]))
{
    // ...
}
```

After this **try** block ends, both **fin** and **fout** will have been closed. If you compare this version of the program to the previous version, you will see that it is much shorter. The ability to streamline source code is a side-benefit of automatic resource management.

There is one other aspect to `try-with-resources` that needs to be mentioned. In general, when a `try` block executes, it is possible that an exception inside the `try` block will lead to another exception that occurs when the resource is closed in a `finally` clause. In the case of a “normal” `try` statement, the original exception is lost, being preempted by the second exception. However, when using `try-with-resources`, the second exception is *suppressed*. It is not, however, lost. Instead, it is added to the list of suppressed exceptions associated with the first exception. The list of suppressed exceptions can be obtained by using the `getSuppressed()` method defined by `Throwable`.

Because of the benefits that the `try-with-resources` statement offers, it will be used by many, but not all, of the example programs in this edition of this book. Some of the examples will still use the traditional approach to closing a resource. There are several reasons for this. First, there is legacy code that still relies on the traditional approach. It is important that all Java programmers be fully versed in, and comfortable with, the traditional approach when maintaining this older code. Second, because not all project development will immediately switch to a new version of the JDK, it is likely that some programmers will continue to work in a pre-JDK 7 environment for a period of time. In such situations, the expanded form of `try` is not available. Finally, there may be cases in which explicitly closing a resource is more appropriate than the automated approach. For these reasons, some of the examples in this book will continue to use the traditional approach, explicitly calling `close()`. In addition to illustrating the traditional technique, these examples can also be compiled and run by all readers in all environments.

REMEMBER A few examples in this book use the traditional approach to closing files as a means of illustrating this technique, which is widely used in legacy code. However, for new code, you will usually want to use the new automated approach supported by the `try-with-resources` statement just described.

Applet Fundamentals

All of the preceding examples in this book have been Java console-based applications. However, these types of applications constitute only one class of Java programs. Another type of program is the applet. As mentioned in Chapter 1, *applets* are small applications that

are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document. After an applet arrives on the client, it has limited access to resources so that it can produce a graphical user interface and run various computations without introducing the risk of viruses or breaching data integrity.

Many of the issues connected with the creation and use of applets are found in Part II, when the **applet** package is examined, and also when Swing is described in Part III. However, the fundamentals connected to the creation of an applet are presented here, because applets are not structured in the same way as the programs that have been used thus far. As you will see, applets differ from console-based applications in several key areas.

Let's begin with the simple applet shown here:

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

public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("A Simple Applet", 20, 20);
    }
}
```

This applet begins with two **import** statements. The first imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user through a GUI framework, not through the console-based I/O classes. One of these frameworks is the AWT, and that is the framework used here to introduce applet programming. The AWT contains very basic support for a window-based, graphical user interface. As you might expect, the AWT is quite large, and a detailed discussion of it is found in Part II of this book. Fortunately, this simple applet makes very limited use of the AWT. (Another commonly used GUI for applets is Swing, but this approach is described later in this book.) The second **import** statement imports the **applet** package, which contains the class **Applet**. Every AWT-based applet that you create must be a subclass (either directly or indirectly) of **Applet**.

The next line in the program declares the class **SimpleApplet**. This class must be declared as **public**, because it will be accessed by code that is outside the program.

Inside **SimpleApplet**, **paint()** is declared. This method is defined by the AWT and must be overridden by the applet. **paint()** is called each time that the applet must redisplay its output. This situation can occur for several reasons. For example, the window in which the applet is running can be overwritten by another window and then uncovered. Or, the applet window can be minimized and then restored. **paint()** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint()** is called. The **paint()** method has one parameter of type **Graphics**. This parameter contains the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

Inside **paint()** is a call to **drawString()**, which is a member of the **Graphics** class. This method outputs a string beginning at the specified X,Y location. It has the following general form:

```
void drawString(String message, int x, int y)
```

Here, *message* is the string to be output beginning at *x,y*. In a Java window, the upper-left corner is location 0,0. The call to **drawString()** in the applet causes the message "A Simple Applet" to be displayed beginning at location 20,20.

Notice that the applet does not have a **main()** method. Unlike Java programs, applets do not begin execution at **main()**. In fact, most applets don't even have a **main()** method. Instead, an applet begins execution when the name of its class is passed to an applet viewer or to a network browser.

After you enter the source code for **SimpleApplet**, compile in the same way that you have been compiling programs. However, running **SimpleApplet** involves a different process. In fact, there are two ways in which you can run an applet:

- Executing the applet within a Java-compatible web browser.
- Using an applet viewer, such as the standard tool, **appletviewer**. An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.

Each of these methods is described next.

One way to execute an applet in a web browser is to write a short HTML text file that contains a tag that loads the applet. At the time of this writing, Oracle recommends using the APPLET tag for this purpose. (The OBJECT tag can also be used. See Chapter 23 for further information regarding applet deployment strategies.) Using APPLET, here is the HTML file that executes **SimpleApplet**:

```
<applet code="SimpleApplet" width=200 height=60>
</applet>
```

The **width** and **height** statements specify the dimensions of the display area used by the applet. (The APPLET tag contains several other options that are examined more closely in Part II.) After you create this file, you can use it to execute the applet.

NOTE Beginning with the release of Java 7, update 21, Java applets must be signed to prevent security warnings when run in a browser. In fact, in some cases, the applet may be prevented from running. Applets stored in the local file system, such as you would create when compiling the examples in this book, are especially sensitive to this change. You may need to adjust the security settings in the Java Control Panel to run a local applet in a browser. At the time of this writing, Oracle recommends against the use of local applets, recommending instead that applets be executed through a web server. Furthermore, it is expected that unsigned local applets will be blocked from execution in the future. In general, for applets that will be distributed via the Internet, such as commercial applications, signing is a virtual necessity. The concepts and techniques required to sign applets (and other types of Java programs) are beyond the scope of this book. However, extensive information is found on Oracle's website. Finally, as mentioned, the easiest way to try the applet examples is to use **appletviewer**.

To execute **SimpleApplet** with an applet viewer, you may also execute the HTML file shown earlier. For example, if the preceding HTML file is called **RunApp.html**, then the following command line will run **SimpleApplet**:

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

However, a more convenient method exists that you can use to speed up testing. Simply include a comment at the head of your Java source code file that contains the

APPLET tag. By doing so, your code is documented with a prototype of the necessary HTML statements, and you can test your compiled applet merely by starting the applet viewer with your Java source code file. If you use this method, the **SimpleApplet** source file looks like this:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/
public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("A Simple Applet", 20, 20);
    }
}
```

With this approach, you can quickly iterate through applet development by using these three steps:

1. Edit a Java source file.
2. Compile your program.
3. Execute the applet viewer, specifying the name of your applet's source file. The applet viewer will encounter the APPLET tag within the comment and execute your applet.

The window produced by **SimpleApplet**, as displayed by the applet viewer, is shown in the following illustration. Of course, the precise appearance of the applet viewer frame may differ based on your environment. For this reason, the screen captures in this book reflect a number of different environments.



While the subject of applets is more fully discussed later in this book, here are the key points that you should remember now:

- Applets do not need a **main()** method.
- Applets must be run under an applet viewer or a Java-compatible browser.
- User I/O is not accomplished with Java's stream I/O classes. Instead, applets use the interface provided by a GUI framework.

The transient and volatile Modifiers

Java defines two interesting type modifiers: **transient** and **volatile**. These modifiers are used to handle somewhat specialized situations.

When an instance variable is declared as **transient**, then its value need not persist when an object is stored. For example:

```
class T {
    transient int a; // will not persist
    int b; // will persist
}
```

Here, if an object of type **T** is written to a persistent storage area, the contents of **a** would not be saved, but the contents of **b** would.

The **volatile** modifier tells the compiler that the variable modified by **volatile** can be changed unexpectedly by other parts of your program. One of these situations involves multithreaded programs. In a multithreaded program, sometimes two or more threads share the same variable. For efficiency considerations, each thread can keep its own, private copy of such a shared variable. The real (or *master*) copy of the variable is updated at various times, such as when a **synchronized** method is entered. While this approach works fine, it may be inefficient at times. In some cases, all that really matters is that the master copy of a variable always reflects its current state. To ensure this, simply specify the variable as **volatile**, which tells the compiler that it must always use the master copy of a **volatile** variable (or, at least, always keep any private copies up-to-date with the master copy, and vice versa). Also, accesses to the master variable must be executed in the precise order in which they are executed on any private copy.

Using instanceof

Sometimes, knowing the type of an object during run time is useful. For example, you might have one thread of execution that generates various types of objects, and another thread that processes these objects. In this situation, it might be useful for the processing thread to know the type of each object when it receives it. Another situation in which knowledge of an object's type at run time is important involves casting. In Java, an invalid cast causes a run-time error. Many invalid casts can be caught at compile time. However, casts involving class hierarchies can produce invalid casts that can be detected only at run time. For example, a superclass called **A** can produce two subclasses, called **B** and **C**. Thus, casting a **B** object into type **A** or casting a **C** object into type **A** is legal, but casting a **B** object into type **C** (or vice versa) isn't legal. Because an object of type **A** can refer to objects of either **B** or **C**, how can you know, at run time, what type of object is actually being referred to before attempting the cast to type **C**? It could be an object of type **A**, **B**, or **C**. If it is an object of type **B**, a run-time exception will be thrown. Java provides the run-time operator **instanceof** to answer this question.

The **instanceof** operator has this general form:

objref instanceof *type*

Here, *objref* is a reference to an instance of a class, and *type* is a class type. If *objref* is of the specified type or can be cast into the specified type, then the **instanceof** operator evaluates to **true**. Otherwise, its result is **false**. Thus, **instanceof** is the means by which your program can obtain run-time type information about an object.

The following program demonstrates **instanceof**:

```
// Demonstrate instanceof operator.
class A {
    int i, j;
}

class B {
    int i, j;
}

class C extends A {
    int k;
}

class D extends A {
    int k;
}

class InstanceOf {
    public static void main(String args[]) {
        A a = new A();
        B b = new B();
        C c = new C();
        D d = new D();
        if(a instanceof A)
            System.out.println("a is instance of A");
        if(b instanceof B)
            System.out.println("b is instance of B");
        if(c instanceof C)
            System.out.println("c is instance of C");
        if(c instanceof A)
            System.out.println("c can be cast to A");

        if(a instanceof C)
            System.out.println("a can be cast to C");

        System.out.println();

        // compare types of derived types
        A ob;

        ob = d; // A reference to d
        System.out.println("ob now refers to d");
        if(ob instanceof D)
            System.out.println("ob is instance of D");

        System.out.println();

        ob = c; // A reference to c
        System.out.println("ob now refers to c");

        if(ob instanceof D)
            System.out.println("ob can be cast to D");
        else
            System.out.println("ob cannot be cast to D");

        if(ob instanceof A)
            System.out.println("ob can be cast to A");

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

```

// all objects can be cast to Object
if(a instanceof Object)
    System.out.println("a may be cast to Object");
if(b instanceof Object)
    System.out.println("b may be cast to Object");
if(c instanceof Object)
    System.out.println("c may be cast to Object");
if(d instanceof Object)
    System.out.println("d may be cast to Object");
}
}

```

The output from this program is shown here:

```

a is instance of A
b is instance of B
c is instance of C
c can be cast to A

ob now refers to d
ob is instance of D

ob now refers to c
ob cannot be cast to D
ob can be cast to A

a may be cast to Object
b may be cast to Object
c may be cast to Object
d may be cast to Object

```

The **instanceof** operator isn't needed by most programs, because, generally, you know the type of object with which you are working. However, it can be very useful when you're writing generalized routines that operate on objects of a complex class hierarchy.

strictfp

With the creation of Java 2, the floating-point computation model was relaxed slightly. Specifically, the new model does not require the truncation of certain intermediate values that occur during a computation. This prevents overflow or underflow in some cases. By modifying a class, a method, or interface with **strictfp**, you ensure that floating-point calculations (and thus all truncations) take place precisely as they did in earlier versions of Java. When a class is modified by **strictfp**, all the methods in the class are also modified by **strictfp** automatically.

For example, the following fragment tells Java to use the original floating-point model for calculations in all methods defined within **MyClass**:

```
strictfp class MyClass { //...
```

Frankly, most programmers never need to use **strictfp**, because it affects only a very small class of problems.

Native Methods

Although it is rare, occasionally you may want to call a subroutine that is written in a language other than Java. Typically, such a subroutine exists as executable code for the CPU and environment in which you are working—that is, native code. For example, you may want to call a native code subroutine to achieve faster execution time. Or, you may want to use a specialized, third-party library, such as a statistical package. However, because Java programs are compiled to bytecode, which is then interpreted (or compiled on-the-fly) by the Java run-time system, it would seem impossible to call a native code subroutine from within your Java program. Fortunately, this conclusion is false. Java provides the **native** keyword, which is used to declare native code methods. Once declared, these methods can be called from inside your Java program just as you call any other Java method.

To declare a native method, precede the method with the **native** modifier, but do not define any body for the method. For example:

```
public native int meth() ;
```

After you declare a native method, you must write the native method and follow a rather complex series of steps to link it with your Java code.

Most native methods are written in C. The mechanism used to integrate C code with a Java program is called the *Java Native Interface (JNI)*. A detailed description of the JNI is beyond the scope of this book, but the approach described here provides sufficient information for simple applications.

NOTE The precise steps that you need to follow will vary between different Java environments. They also depend on the language that you are using to implement the native method. The following discussion assumes a Windows environment. The language used to implement the native method is C. Also, the approach shown here uses a dynamically linked library, but beginning with JDK 8, it is possible to create a statically linked library.

The easiest way to understand the process is to work through an example. To begin, enter the following short program, which uses a **native** method called **test()**:

```
// A simple example that uses a native method.
public class NativeDemo {
    int i;
    public static void main(String args[]) {
        NativeDemo ob = new NativeDemo();

        ob.i = 10;
        System.out.println("This is ob.i before the native method:" +
                           ob.i);
        ob.test(); // call a native method
        System.out.println("This is ob.i after the native method:" +
                           ob.i);
    }

    // declare native method
    public native void test() ;

    // load DLL that contains static method
```

```

    static {
        System.loadLibrary("NativeDemo");
    }
}

```

Notice that the **test()** method is declared as **native** and has no body. This is the method that we will implement in C shortly. Also notice the **static** block. As explained earlier in this book, a **static** block is executed only once, when your program begins execution (or, more precisely, when its class is first loaded). In this case, it is used to load the dynamic link library that contains the native implementation of **test()**. (You will see how to create this library soon.)

The library is loaded by the **loadLibrary()** method, which is part of the **System** class. This is its general form:

```
static void loadLibrary(String filename)
```

Here, *filename* is a string that specifies the name of the file that holds the library. For the Windows environment, this file is assumed to have the .DLL extension.

After you enter the program, compile it to produce **NativeDemo.class**. Next, you must use **javah.exe** to produce one file: **NativeDemo.h**. (**javah.exe** is included in the JDK.) You will include **NativeDemo.h** in your implementation of **test()**. To produce **NativeDemo.h**, use the following command:

```
javah -jni NativeDemo
```

This command produces a header file called **NativeDemo.h**. This file must be included in the C file that implements **test()**. The output produced by this command is shown here:

```

/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class NativeDemo */

#ifndef _Included_NativeDemo
#define _Included_NativeDemo
#ifndef __cplusplus
extern "C" {
#endif
/*
 * Class:      NativeDemo
 * Method:     test
 * Signature:  ()V
 */
JNIEXPORT void JNICALL Java_NativeDemo_test
    (JNIEnv *, jobject);

#endif /* __cplusplus
#endif
#endif

```

Pay special attention to the following line, which defines the prototype for the **test()** function that you will create:

```
JNICALL void Java_NativeDemo_test(JNIEnv *, jobject);
```

Notice that the name of the function is `Java_NativeDemo_test()`. You must use this as the name of the native function that you implement. That is, instead of creating a C function called `test()`, you will create one called `Java_NativeDemo_test()`. The `NativeDemo` component of the prefix is added because it identifies the `test()` method as being part of the `NativeDemo` class. Remember, another class may define its own native `test()` method that is completely different from the one declared by `NativeDemo`. Including the class name in the prefix provides a way to differentiate between differing versions. As a general rule, native functions will be given a name whose prefix includes the name of the class in which they are declared.

After producing the necessary header file, you can write your implementation of `test()` and store it in a file named `NativeDemo.c`:

```
/* This file contains the C version of the
   test() method.
*/
#include <jni.h>
#include "NativeDemo.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *env, jobject obj)
{
    jclass cls;
    jfieldID fid;
    jint i;

    printf("Starting the native method.\n");
    cls = (*env)->GetObjectClass(env, obj);
    fid = (*env)->GetFieldID(env, cls, "i", "I");

    if(fid == 0) {
        printf("Could not get field id.\n");
        return;
    }
    i = (*env)->GetIntField(env, obj, fid);
    printf("i = %d\n", i);
    (*env)->SetIntField(env, obj, fid, 2*i);
    printf("Ending the native method.\n");
}
```

Notice that this file includes `jni.h`, which contains interfacing information. This file is provided by your Java compiler. The header file `NativeDemo.h` was created by `javadoc` earlier.

In this function, the `GetObjectClass()` method is used to obtain a C structure that has information about the class `NativeDemo`. The `GetFieldID()` method returns a C structure with information about the field named "i" for the class. `GetIntField()` retrieves the original value of that field. `SetIntField()` stores an updated value in that field. (See the file `jni.h` for additional methods that handle other types of data.)

After creating **NativeDemo.c**, you must compile it and create a DLL. To do this by using the Microsoft C/C++ compiler, use the following command line. (You might need to specify the path to **jni.h** and its subordinate file **jni_md.h**.)

```
Cl /LD NativeDemo.c
```

This produces a file called **NativeDemo.dll**. Once this is done, you can execute the Java program, which will produce the following output:

```
This is ob.i before the native method: 10
Starting the native method.
i = 10
Ending the native method.
This is ob.i after the native method: 20
```

Problems with Native Methods

Native methods seem to offer great promise, because they enable you to gain access to an existing base of library routines, and they offer the possibility of faster run-time execution. But native methods also introduce two significant problems:

- **Potential security risk** Because a native method executes actual machine code, it can gain access to any part of the host system. That is, native code is not confined to the Java execution environment. This could allow a virus infection, for example. For this reason, unsigned applets cannot use native methods. Also, the loading of DLLs can be restricted, and their loading is subject to the approval of the security manager.
- **Loss of portability** Because the native code is contained in a DLL, it must be present on the machine that is executing the Java program. Further, because each native method is CPU- and operating system-dependent, each DLL is inherently nonportable. Thus, a Java application that uses native methods will be able to run only on a machine for which a compatible DLL has been installed.

The use of native methods should be restricted, because they render your Java programs nonportable and pose significant security risks.

Using assert

Another relatively new addition to Java is the keyword **assert**. It is used during program development to create an *assertion*, which is a condition that should be true during the execution of the program. For example, you might have a method that should always return a positive integer value. You might test this by asserting that the return value is greater than zero using an **assert** statement. At run time, if the condition is true, no other action takes place. However, if the condition is false, then an **AssertionError** is thrown. Assertions are often used during testing to verify that some expected condition is actually met. They are not usually used for released code.

The **assert** keyword has two forms. The first is shown here:

```
assert condition;
```

Here, *condition* is an expression that must evaluate to a Boolean result. If the result is true, then the assertion is true and no other action takes place. If the condition is false, then the assertion fails and a default **AssertionError** object is thrown.

The second form of **assert** is shown here:

```
assert condition: expr;
```

In this version, *expr* is a value that is passed to the **AssertionError** constructor. This value is converted to its string format and displayed if an assertion fails. Typically, you will specify a string for *expr*, but any non-void expression is allowed as long as it defines a reasonable string conversion.

Here is an example that uses **assert**. It verifies that the return value of **getnum()** is positive.

```
// Demonstrate assert.
class AssertDemo {
    static int val = 3;

    // Return an integer.
    static int getnum() {
        return val--;
    }

    public static void main(String args[])
    {
        int n;

        for(int i=0; i < 10; i++) {
            n = getnum();

            assert n > 0; // will fail when n is 0
            System.out.println("n is " + n);
        }
    }
}
```

To enable assertion checking at run time, you must specify the **-ea** option. For example, to enable assertions for **AssertDemo**, execute it using this line:

```
java -ea AssertDemo
```

After compiling and running as just described, the program creates the following output:

```
n is 3
n is 2
n is 1
Exception in thread "main" java.lang.AssertionError
at AssertDemo.main(AssertDemo.java:17)
```

In `main()`, repeated calls are made to the method `getnum()`, which returns an integer value. The return value of `getnum()` is assigned to `n` and then tested using this `assert` statement:

```
assert n > 0; // will fail when n is 0
```

This statement will fail when `n` equals 0, which it will after the fourth call. When this happens, an exception is thrown.

As explained, you can specify the message displayed when an assertion fails. For example, if you substitute

```
assert n > 0 : "n is negative!";
```

for the assertion in the preceding program, then the following output will be generated:

```
n is 3
n is 2
n is 1
Exception in thread "main" java.lang.AssertionError: n is
negative!
        at AssertDemo.main(AssertDemo.java:17)
```

One important point to understand about assertions is that you must not rely on them to perform any action actually required by the program. The reason is that normally, released code will be run with assertions disabled. For example, consider this variation of the preceding program:

```
// A poor way to use assert!!!
class AssertDemo {
    // get a random number generator
    static int val = 3;

    // Return an integer.
    static int getnum() {
        return val--;
    }

    public static void main(String args[])
    {
        int n = 0;

        for(int i=0; i < 10; i++) {

            assert (n = getnum()) > 0; // This is not a good idea!

            System.out.println("n is " + n);
        }
    }
}
```

In this version of the program, the call to `getnum()` is moved inside the `assert` statement. Although this works fine if assertions are enabled, it will cause a malfunction when assertions are disabled, because the call to `getnum()` will never be executed! In fact, `n` must now be

initialized, because the compiler will recognize that it might not be assigned a value by the **assert** statement.

Assertions are a good addition to Java because they streamline the type of error checking that is common during development. For example, prior to **assert**, if you wanted to verify that **n** was positive in the preceding program, you had to use a sequence of code similar to this:

```
if(n < 0) {
    System.out.println("n is negative!");
    return; // or throw an exception
}
```

With **assert**, you need only one line of code. Furthermore, you don't have to remove the **assert** statements from your released code.

Assertion Enabling and Disabling Options

When executing code, you can disable all assertions by using the **-da** option. You can enable or disable a specific package (and all of its subpackages) by specifying its name followed by three periods after the **-ea** or **-da** option. For example, to enable assertions in a package called **MyPack**, use

```
-ea :MyPack...
```

To disable assertions in **MyPack**, use

```
-da :MyPack...
```

You can also specify a class with the **-ea** or **-da** option. For example, this enables **AssertDemo** individually:

```
-ea :AssertDemo
```

Static Import

Java includes a feature called *static import* that expands the capabilities of the **import** keyword. By following **import** with the keyword **static**, an **import** statement can be used to import the static members of a class or interface. When using static import, it is possible to refer to static members directly by their names, without having to qualify them with the name of their class. This simplifies and shortens the syntax required to use a static member.

To understand the usefulness of static import, let's begin with an example that does *not* use it. The following program computes the hypotenuse of a right triangle. It uses two static methods from Java's built-in math class **Math**, which is part of **java.lang**. The first is **Math.pow()**, which returns a value raised to a specified power. The second is **Math.sqrt()**, which returns the square root of its argument.

```
// Compute the hypotenuse of a right triangle.
class Hypot {
    public static void main(String args[]) {
        double side1, side2;
```

```

        double hypot;
        side1 = 3.0;
        side2 = 4.0;

        // Notice how sqrt() and pow() must be qualified by
        // their class name, which is Math.
        hypot = Math.sqrt(Math.pow(side1, 2) +
                          Math.pow(side2, 2));

        System.out.println("Given sides of lengths " +
                           side1 + " and " + side2 +
                           " the hypotenuse is " +
                           hypot);
    }
}

```

Because **pow()** and **sqrt()** are static methods, they must be called through the use of their class' name, **Math**. This results in a somewhat unwieldy hypotenuse calculation:

```
hypot = Math.sqrt(Math.pow(side1, 2) +
                  Math.pow(side2, 2));
```

As this simple example illustrates, having to specify the class name each time **pow()** or **sqrt()** (or any of Java's other math methods, such as **sin()**, **cos()**, and **tan()**) is used can grow tedious.

You can eliminate the tedium of specifying the class name through the use of static import, as shown in the following version of the preceding program:

```

// Use static import to bring sqrt() and pow() into view.
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

// Compute the hypotenuse of a right triangle.
class Hypot {
    public static void main(String args[]) {
        double side1, side2;
        double hypot;

        side1 = 3.0;
        side2 = 4.0;

        // Here, sqrt() and pow() can be called by themselves,
        // without their class name.
        hypot = sqrt(pow(side1, 2) + pow(side2, 2));

        System.out.println("Given sides of lengths " +
                           side1 + " and " + side2 +
                           " the hypotenuse is " +
                           hypot);
    }
}

```

In this version, the names **sqrt** and **pow** are brought into view by these static import statements:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
```

After these statements, it is no longer necessary to qualify **sqrt()** or **pow()** with their class name. Therefore, the hypotenuse calculation can more conveniently be specified, as shown here:

```
hypot = sqrt(pow(side1, 2) + pow(side2, 2));
```

As you can see, this form is considerably more readable.

There are two general forms of the **import static** statement. The first, which is used by the preceding example, brings into view a single name. Its general form is shown here:

```
import static pkg.type-name.static-member-name;
```

Here, *type-name* is the name of a class or interface that contains the desired static member. Its full package name is specified by *pkg*. The name of the member is specified by *static-member-name*.

The second form of static import imports all static members of a given class or interface. Its general form is shown here:

```
import static pkg.type-name.*;
```

If you will be using many static methods or fields defined by a class, then this form lets you bring them into view without having to specify each individually. Therefore, the preceding program could have used this single **import** statement to bring both **pow()** and **sqrt()** (and *all other* static members of **Math**) into view:

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

Of course, static import is not limited just to the **Math** class or just to methods. For example, this brings the static field **System.out** into view:

```
import static java.lang.System.out;
```

After this statement, you can output to the console without having to qualify **out** with **System**, as shown here:

```
out.println("After importing System.out, you can use out directly.");
```

Whether importing **System.out** as just shown is a good idea is subject to debate. Although it does shorten the statement, it is no longer instantly clear to anyone reading the program that the **out** being referred to is **System.out**.

One other point: in addition to importing the static members of classes and interfaces defined by the Java API, you can also use static import to import the static members of classes and interfaces that you create.

As convenient as static import can be, it is important not to abuse it. Remember, the reason that Java organizes its libraries into packages is to avoid namespace collisions. When you import static members, you are bringing those members into the global namespace. Thus, you are increasing the potential for namespace conflicts and for the inadvertent hiding of other names. If you are using a static member once or twice in the program, it's best not to import it. Also, some static names, such as `System.out`, are so recognizable that you might not want to import them. Static import is designed for those situations in which you are using a static member repeatedly, such as when performing a series of mathematical computations. In essence, you should use, but not abuse, this feature.

Invoking Overloaded Constructors Through `this()`

When working with overloaded constructors, it is sometimes useful for one constructor to invoke another. In Java, this is accomplished by using another form of the `this` keyword. The general form is shown here:

`this(arg-list)`

When `this()` is executed, the overloaded constructor that matches the parameter list specified by *arg-list* is executed first. Then, if there are any statements inside the original constructor, they are executed. The call to `this()` must be the first statement within the constructor.

To understand how `this()` can be used, let's work through a short example. First, consider the following class that *does not* use `this()`:

```
class MyClass {  
    int a;  
    int b;  
  
    // initialize a and b individually  
    MyClass(int i, int j) {  
        a = i;  
        b = j;  
    }  
  
    // initialize a and b to the same value  
    MyClass(int i) {  
        a = i;  
        b = i;  
    }  
  
    // give a and b default values of 0  
    MyClass() {  
        a = 0;  
        b = 0;  
    }  
}
```

This class contains three constructors, each of which initializes the values of **a** and **b**. The first is passed individual values for **a** and **b**. The second is passed just one value, which is assigned to both **a** and **b**. The third gives **a** and **b** default values of zero.

By using **this()**, it is possible to rewrite **MyClass** as shown here:

```
class MyClass {
    int a;
    int b;

    // initialize a and b individually
    MyClass(int i, int j) {
        a = i;
        b = j;
    }

    // initialize a and b to the same value
    MyClass(int i) {
        this(i, i); // invokes MyClass(i, i)
    }

    // give a and b default values of 0
    MyClass() {
        this(0); // invokes MyClass(0)
    }
}
```

In this version of **MyClass**, the only constructor that actually assigns values to the **a** and **b** fields is **MyClass(int, int)**. The other two constructors simply invoke that constructor (either directly or indirectly) through **this()**. For example, consider what happens when this statement executes:

```
MyClass mc = new MyClass(8);
```

The call to **MyClass(8)** causes **this(8, 8)** to be executed, which translates into a call to **MyClass(8, 8)**, because this is the version of the **MyClass** constructor whose parameter list matches the arguments passed via **this()**. Now, consider the following statement, which uses the default constructor:

```
MyClass mc2 = new MyClass();
```

In this case, **this(0)** is called. This causes **MyClass(0)** to be invoked because it is the constructor with the matching parameter list. Of course, **MyClass(0)** then calls **MyClass(0,0)** as just described.

One reason why invoking overloaded constructors through **this()** can be useful is that it can prevent the unnecessary duplication of code. In many cases, reducing duplicate code decreases the time it takes to load your class because often the object code is smaller. This is especially important for programs delivered via the Internet in which load times are an issue. Using **this()** can also help structure your code when constructors contain a large amount of duplicate code.

However, you need to be careful. Constructors that call **this()** will execute a bit slower than those that contain all of their initialization code inline. This is because the call and return mechanism used when the second constructor is invoked adds overhead. If your class will be used to create only a handful of objects, or if the constructors in the class that call **this()** will be seldom used, then this decrease in run-time performance is probably

insignificant. However, if your class will be used to create a large number of objects (on the order of thousands) during program execution, then the negative impact of the increased overhead could be meaningful. Because object creation affects all users of your class, there will be cases in which you must carefully weigh the benefits of faster load time against the increased time it takes to create an object.

Here is another consideration: for very short constructors, such as those used by `MyClass`, there is often little difference in the size of the object code whether `this()` is used or not. (Actually, there are cases in which no reduction in the size of the object code is achieved.) This is because the bytecode that sets up and returns from the call to `this()` adds instructions to the object file. Therefore, in these types of situations, even though duplicate code is eliminated, using `this()` will not obtain significant savings in terms of load time. However, the added cost in terms of overhead to each object's construction will still be incurred. Therefore, `this()` is most applicable to constructors that contain large amounts of initialization code, not those that simply set the value of a handful of fields.

There are two restrictions you need to keep in mind when using `this()`. First, you cannot use any instance variable of the constructor's class in a call to `this()`. Second, you cannot use `super()` and `this()` in the same constructor because each must be the first statement in the constructor.

Compact API Profiles

JDK 8 adds a feature that organizes subsets of the API library into what are called *compact profiles*. These are called `compact1`, `compact2`, and `compact3`. Each profile contains a subset of the library. Furthermore, `compact2` includes all of `compact1`, and `compact3` includes all of `compact2`. Thus, each profile builds on the previous one. The advantage of the compact profiles is that an application that does not require the full library need not download it. Using a compact profile reduces the size of the library, thus enabling some types of Java applications to run on devices that could not otherwise support the entire Java API. The use of a compact profile can also reduce the time it takes to load a program. The Java API documentation indicates to which (if any) profile each API element belongs.

When compiling a program, you can determine if a program uses only APIs defined by a compact profile by using the `-profile` option. Here is its general form:

```
javac -profile profileName programName
```

Here, *profileName* specifies the profile, which must be `compact1`, `compact2`, or `compact3`. For example:

```
javac -profile compact2 Test.java
```

Here, the `compact2` profile is specified. If `Test.java` contains an API that is not part of `compact2`, then a compilation error will result.

CHAPTER

14

Generics

Since the original 1.0 release in 1995, many new features have been added to Java. One that has had a profound impact is *generics*. Introduced by JDK 5, generics changed Java in two important ways. First, it added a new syntactical element to the language. Second, it caused changes to many of the classes and methods in the core API. Today, generics are an integral part of Java programming, and a solid understanding of this important feature is required. It is examined here in detail.

Through the use of generics, it is possible to create classes, interfaces, and methods that will work in a type-safe manner with various kinds of data. Many algorithms are logically the same no matter what type of data they are being applied to. For example, the mechanism that supports a stack is the same whether that stack is storing items of type **Integer**, **String**, **Object**, or **Thread**. With generics, you can define an algorithm once, independently of any specific type of data, and then apply that algorithm to a wide variety of data types without any additional effort. The expressive power generics added to the language fundamentally changed the way that Java code is written.

Perhaps the one feature of Java that has been most significantly affected by generics is the *Collections Framework*. The Collections Framework is part of the Java API and is described in detail in Chapter 18, but a brief mention is useful now. A *collection* is a group of objects. The Collections Framework defines several classes, such as lists and maps, that manage collections. The collection classes have always been able to work with any type of object. The benefit that generics added is that the collection classes can now be used with complete type safety. Thus, in addition to being a powerful language element on its own, generics also enabled an existing feature to be substantially improved. This is another reason why generics were such an important addition to Java.

This chapter describes the syntax, theory, and use of generics. It also shows how generics provide type safety for some previously difficult cases. Once you have completed this chapter, you will want to examine Chapter 18, which covers the Collections Framework. There you will find many examples of generics at work.

What Are Generics?

At its core, the term *generics* means *parameterized types*. Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter. Using generics, it is possible to create a single class, for example, that automatically works with different types of data. A class, interface, or method that operates on a parameterized type is called *generic*, as in *generic class* or *generic method*.

It is important to understand that Java has always given you the ability to create generalized classes, interfaces, and methods by operating through references of type **Object**. Because **Object** is the superclass of all other classes, an **Object** reference can refer to any type object. Thus, in pre-generics code, generalized classes, interfaces, and methods used **Object** references to operate on various types of objects. The problem was that they could not do so with type safety.

Generics added the type safety that was lacking. They also streamlined the process, because it is no longer necessary to explicitly employ casts to translate between **Object** and the type of data that is actually being operated upon. With generics, all casts are automatic and implicit. Thus, generics expanded your ability to reuse code and let you do so safely and easily.

NOTE A Warning to C++ Programmers: Although generics are similar to templates in C++, they are not the same. There are some fundamental differences between the two approaches to generic types. If you have a background in C++, it is important not to jump to conclusions about how generics work in Java.

A Simple Generics Example

Let's begin with a simple example of a generic class. The following program defines two classes. The first is the generic class **Gen**, and the second is **GenDemo**, which uses **Gen**.

```
// A simple generic class.  
// Here, T is a type parameter that  
// will be replaced by a real type  
// when an object of type Gen is created.  
class Gen<T> {  
    T ob; // declare an object of type T  
  
    // Pass the constructor a reference to  
    // an object of type T.  
    Gen(T o) {  
        ob = o;  
    }  
  
    // Return ob.  
    T getob() {  
        return ob;  
    }  
  
    // Show type of T.  
}
```

```

void showType() {
    System.out.println("Type of T is " +
                       ob.getClass().getName());
}
}

// Demonstrate the generic class.
class GenDemo {
    public static void main(String args[]) {
        // Create a Gen reference for Integers.
        Gen<Integer> iOb;

        // Create a Gen<Integer> object and assign its
        // reference to iOb. Notice the use of autoboxing
        // to encapsulate the value 88 within an Integer object.
        iOb = new Gen<Integer>(88);

        // Show the type of data used by iOb.
        iOb.showType();

        // Get the value in iOb. Notice that
        // no cast is needed.
        int v = iOb.getob();
        System.out.println("value: " + v);

        System.out.println();

        // Create a Gen object for Strings.
        Gen<String> strOb = new Gen<String> ("Generics Test");

        // Show the type of data used by strOb.
        strOb.showType();

        // Get the value of strOb. Again, notice
        // that no cast is needed.
        String str = strOb.getob();
        System.out.println("value: " + str);
    }
}

```

The output produced by the program is shown here:

```
Type of T is java.lang.Integer
value: 88
```

```
Type of T is java.lang.String
value: Generics Test
```

Let's examine this program carefully.

First, notice how **Gen** is declared by the following line:

```
class Gen<T> {
```

Here, **T** is the name of a *type parameter*. This name is used as a placeholder for the actual type that will be passed to **Gen** when an object is created. Thus, **T** is used within **Gen** whenever the type parameter is needed. Notice that **T** is contained within **<>**. This syntax can be generalized. Whenever a type parameter is being declared, it is specified within angle brackets. Because **Gen** uses a type parameter, **Gen** is a generic class, which is also called a *parameterized type*.

Next, **T** is used to declare an object called **ob**, as shown here:

```
T ob; // declare an object of type T
```

As explained, **T** is a placeholder for the actual type that will be specified when a **Gen** object is created. Thus, **ob** will be an object of the type passed to **T**. For example, if type **String** is passed to **T**, then in that instance, **ob** will be of type **String**.

Now consider **Gen**'s constructor:

```
Gen(T o) {
    ob = o;
}
```

Notice that its parameter, **o**, is of type **T**. This means that the actual type of **o** is determined by the type passed to **T** when a **Gen** object is created. Also, because both the parameter **o** and the member variable **ob** are of type **T**, they will both be of the same actual type when a **Gen** object is created.

The type parameter **T** can also be used to specify the return type of a method, as is the case with the **getob()** method, shown here:

```
T getob() {
    return ob;
}
```

Because **ob** is also of type **T**, its type is compatible with the return type specified by **getob()**.

The **showType()** method displays the type of **T** by calling **getName()** on the **Class** object returned by the call to **getClass()** on **ob**. The **getClass()** method is defined by **Object** and is thus a member of all class types. It returns a **Class** object that corresponds to the type of the class of the object on which it is called. **Class** defines the **getName()** method, which returns a string representation of the class name.

The **GenDemo** class demonstrates the generic **Gen** class. It first creates a version of **Gen** for integers, as shown here:

```
Gen<Integer> iob;
```

Look closely at this declaration. First, notice that the type **Integer** is specified within the angle brackets after **Gen**. In this case, **Integer** is a *type argument* that is passed to **Gen**'s type parameter, **T**. This effectively creates a version of **Gen** in which all references to **T** are translated into references to **Integer**. Thus, for this declaration, **ob** is of type **Integer**, and the return type of **getob()** is of type **Integer**.

Before moving on, it's necessary to state that the Java compiler does not actually create different versions of **Gen**, or of any other generic class. Although it's helpful to think in these terms, it is not what actually happens. Instead, the compiler removes all generic type information, substituting the necessary casts, to make your code *behave as if* a specific version of **Gen** were created. Thus, there is really only one version of **Gen** that actually exists in your program. The process of removing generic type information is called *erasure*, and we will return to this topic later in this chapter.

The next line assigns to **iOb** a reference to an instance of an **Integer** version of the **Gen** class:

```
iOb = new Gen<Integer>(88);
```

Notice that when the **Gen** constructor is called, the type argument **Integer** is also specified. This is because the type of the object (in this case **iOb**) to which the reference is being assigned is of type **Gen<Integer>**. Thus, the reference returned by **new** must also be of type **Gen<Integer>**. If it isn't, a compile-time error will result. For example, the following assignment will cause a compile-time error:

```
iOb = new Gen<Double>(88.0); // Error!
```

Because **iOb** is of type **Gen<Integer>**, it can't be used to refer to an object of **Gen<Double>**. This type checking is one of the main benefits of generics because it ensures type safety.

NOTE As you will see later in this chapter, beginning with JDK 7, it is possible to shorten the syntax used to create an instance of a generic class. In the interest of clarity, we will use the full syntax at this time.

As the comments in the program state, the assignment

```
iOb = new Gen<Integer>(88);
```

makes use of autoboxing to encapsulate the value 88, which is an **int**, into an **Integer**. This works because **Gen<Integer>** creates a constructor that takes an **Integer** argument. Because an **Integer** is expected, Java will automatically box 88 inside one. Of course, the assignment could also have been written explicitly, like this:

```
iOb = new Gen<Integer>(new Integer(88));
```

However, there would be no benefit to using this version.

The program then displays the type of **ob** within **iOb**, which is **Integer**. Next, the program obtains the value of **ob** by use of the following line:

```
int v = iOb.getob();
```

Because the return type of **getob()** is **T**, which was replaced by **Integer** when **iOb** was declared, the return type of **getob()** is also **Integer**, which unboxes into **int** when assigned to **v** (which is an **int**). Thus, there is no need to cast the return type of **getob()** to **Integer**.

Of course, it's not necessary to use the auto-unboxing feature. The preceding line could have been written like this, too:

```
int v = iOb.getob().intValue();
```

However, the auto-unboxing feature makes the code more compact.

Next, **GenDemo** declares an object of type **Gen<String>**:

```
Gen<String> strOb = new Gen<String>("Generics Test");
```

Because the type argument is **String**, **String** is substituted for **T** inside **Gen**. This creates (conceptually) a **String** version of **Gen**, as the remaining lines in the program demonstrate.

Generics Work Only with Reference Types

When declaring an instance of a generic type, the type argument passed to the type parameter must be a reference type. You cannot use a primitive type, such as **int** or **char**. For example, with **Gen**, it is possible to pass any class type to **T**, but you cannot pass a primitive type to a type parameter. Therefore, the following declaration is illegal:

```
Gen<int> intOb = new Gen<int>(53); // Error, can't use primitive type
```

Of course, not being able to specify a primitive type is not a serious restriction because you can use the type wrappers (as the preceding example did) to encapsulate a primitive type. Further, Java's autoboxing and auto-unboxing mechanism makes the use of the type wrapper transparent.

Generic Types Differ Based on Their Type Arguments

A key point to understand about generic types is that a reference of one specific version of a generic type is not type compatible with another version of the same generic type. For example, assuming the program just shown, the following line of code is in error and will not compile:

```
iOb = strOb; // Wrong!
```

Even though both **iOb** and **strOb** are of type **Gen<T>**, they are references to different types because their type parameters differ. This is part of the way that generics add type safety and prevent errors.

How Generics Improve Type Safety

At this point, you might be asking yourself the following question: Given that the same functionality found in the generic **Gen** class can be achieved without generics, by simply specifying **Object** as the data type and employing the proper casts, what is the benefit of making **Gen** generic? The answer is that generics automatically ensure the type safety of all operations involving **Gen**. In the process, they eliminate the need for you to enter casts and to type-check code by hand.

To understand the benefits of generics, first consider the following program that creates a non-generic equivalent of **Gen**:

```
// NonGen is functionally equivalent to Gen
// but does not use generics.
class NonGen {
    Object ob; // ob is now of type Object

    // Pass the constructor a reference to
    // an object of type Object
    NonGen(Object o) {
        ob = o;
    }

    // Return type Object.
    Object getob() {
        return ob;
    }

    // Show type of ob.
    void showType() {
        System.out.println("Type of ob is " +
                           ob.getClass().getName());
    }
}

// Demonstrate the non-generic class.
class NonGenDemo {
    public static void main(String args[]) {
        NonGen iOb;

        // Create NonGen Object and store
        // an Integer in it. Autoboxing still occurs.
        iOb = new NonGen(88);

        // Show the type of data used by iOb.
        iOb.showType();

        // Get the value of iOb.
        // This time, a cast is necessary.
        int v = (Integer) iOb.getob();
        System.out.println("value: " + v);

        System.out.println();

        // Create another NonGen object and
        // store a String in it.
        NonGen strOb = new NonGen("Non-Generic Test");

        // Show the type of data used by strOb.
        strOb.showType();

        // Get the value of strOb.
        // Again, notice that a cast is necessary.
```

```

String str = (String) strOb.getob();
System.out.println("value: " + str);

// This compiles, but is conceptually wrong!
iOb = strOb;
v = (Integer) iOb.getob(); // run-time error!
}
}

```

There are several things of interest in this version. First, notice that **NonGen** replaces all uses of **T** with **Object**. This makes **NonGen** able to store any type of object, as can the generic version. However, it also prevents the Java compiler from having any real knowledge about the type of data actually stored in **NonGen**, which is bad for two reasons. First, explicit casts must be employed to retrieve the stored data. Second, many kinds of type mismatch errors cannot be found until run time. Let's look closely at each problem.

Notice this line:

```
int v = (Integer) iOb.getob();
```

Because the return type of **getob()** is **Object**, the cast to **Integer** is necessary to enable that value to be auto-unboxed and stored in **v**. If you remove the cast, the program will not compile. With the generic version, this cast was implicit. In the non-generic version, the cast must be explicit. This is not only an inconvenience, but also a potential source of error.

Now, consider the following sequence from near the end of the program:

```

// This compiles, but is conceptually wrong!
iOb = strOb;
v = (Integer) iOb.getob(); // run-time error!

```

Here, **strOb** is assigned to **iOb**. However, **strOb** refers to an object that contains a string, not an integer. This assignment is syntactically valid because all **NonGen** references are the same, and any **NonGen** reference can refer to any other **NonGen** object. However, the statement is semantically wrong, as the next line shows. Here, the return type of **getob()** is cast to **Integer**, and then an attempt is made to assign this value to **v**. The trouble is that **iOb** now refers to an object that stores a **String**, not an **Integer**. Unfortunately, without the use of generics, the Java compiler has no way to know this. Instead, a run-time exception occurs when the cast to **Integer** is attempted. As you know, it is extremely bad to have run-time exceptions occur in your code!

The preceding sequence can't occur when generics are used. If this sequence were attempted in the generic version of the program, the compiler would catch it and report an error, thus preventing a serious bug that results in a run-time exception. The ability to create type-safe code in which type-mismatch errors are caught at compile time is a key advantage of generics. Although using **Object** references to create “generic” code has always been possible, that code was not type safe, and its misuse could result in run-time exceptions. Generics prevent this from occurring. In essence, through generics, run-time errors are converted into compile-time errors. This is a major advantage.

A Generic Class with Two Type Parameters

You can declare more than one type parameter in a generic type. To specify two or more type parameters, simply use a comma-separated list. For example, the following **TwoGen** class is a variation of the **Gen** class that has two type parameters:

```
// A simple generic class with two type
// parameters: T and V.
class TwoGen<T, V> {
    T ob1;
    V ob2;

    // Pass the constructor a reference to
    // an object of type T and an object of type V.
    TwoGen(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }

    // Show types of T and V.
    void showTypes() {
        System.out.println("Type of T is " +
                           ob1.getClass().getName());

        System.out.println("Type of V is " +
                           ob2.getClass().getName());
    }

    T getob1() {
        return ob1;
    }

    V getob2() {
        return ob2;
    }
}

// Demonstrate TwoGen.
class SimpGen {
    public static void main(String args[]) {

        TwoGen<Integer, String> tgObj =
            new TwoGen<Integer, String>(88, "Generics");

        // Show the types.
        tgObj.showTypes();

        // Obtain and show values.
        int v = tgObj.getob1();
        System.out.println("value: " + v);

        String str = tgObj.getob2();
        System.out.println("value: " + str);
    }
}
```

The output from this program is shown here:

```
Type of T is java.lang.Integer
Type of V is java.lang.String
value: 88
value: Generics
```

Notice how **TwoGen** is declared:

```
class TwoGen<T, V> {
```

It specifies two type parameters: **T** and **V**, separated by a comma. Because it has two type parameters, two type arguments must be passed to **TwoGen** when an object is created, as shown next:

```
TwoGen<Integer, String> tgObj =
    new TwoGen<Integer, String>(88, "Generics");
```

In this case, **Integer** is substituted for **T**, and **String** is substituted for **V**.

Although the two type arguments differ in this example, it is possible for both types to be the same. For example, the following line of code is valid:

```
TwoGen<String, String> x = new TwoGen<String, String> ("A", "B");
```

In this case, both **T** and **V** would be of type **String**. Of course, if the type arguments were always the same, then two type parameters would be unnecessary.

The General Form of a Generic Class

The generics syntax shown in the preceding examples can be generalized. Here is the syntax for declaring a generic class:

```
class class-name<type-param-list> { // ...
```

Here is the full syntax for declaring a reference to a generic class and instance creation:

```
class-name<type-arg-list> var-name =
    new class-name<type-arg-list>(cons-arg-list);
```

Bounded Types

In the preceding examples, the type parameters could be replaced by any class type. This is fine for many purposes, but sometimes it is useful to limit the types that can be passed to a type parameter. For example, assume that you want to create a generic class that contains a method that returns the average of an array of numbers. Furthermore, you want to use the class to obtain the average of an array of any type of number, including integers, **floats**, and **doubles**. Thus, you want to specify the type of the numbers generically, using a type parameter. To create such a class, you might try something like this:

```
// Stats attempts (unsuccessfully) to
// create a generic class that can compute
```

```

// the average of an array of numbers of
// any given type.
//
// The class contains an error!
class Stats<T> {
    T[] nums; // nums is an array of type T

    // Pass the constructor a reference to
    // an array of type T.
    Stats(T[] o) {
        nums = o;
    }

    // Return type double in all cases.
    double average() {
        double sum = 0.0;
        for(int i=0; i < nums.length; i++)
            sum += nums[i].doubleValue(); // Error!!!

        return sum / nums.length;
    }
}

```

In **Stats**, the **average()** method attempts to obtain the **double** version of each number in the **nums** array by calling **doubleValue()**. Because all numeric classes, such as **Integer** and **Double**, are subclasses of **Number**, and **Number** defines the **doubleValue()** method, this method is available to all numeric wrapper classes. The trouble is that the compiler has no way to know that you are intending to create **Stats** objects using only numeric types. Thus, when you try to compile **Stats**, an error is reported that indicates that the **doubleValue()** method is unknown. To solve this problem, you need some way to tell the compiler that you intend to pass only numeric types to **T**. Furthermore, you need some way to *ensure* that *only* numeric types are actually passed.

To handle such situations, Java provides *bounded types*. When specifying a type parameter, you can create an upper bound that declares the superclass from which all type arguments must be derived. This is accomplished through the use of an **extends** clause when specifying the type parameter, as shown here:

<Textends superclass>

This specifies that **T** can only be replaced by *superclass*, or subclasses of *superclass*. Thus, *superclass* defines an inclusive, upper limit.

You can use an upper bound to fix the **Stats** class shown earlier by specifying **Number** as an upper bound, as shown here:

```

// In this version of Stats, the type argument for
// T must be either Number, or a class derived
// from Number.
class Stats<T extends Number> {
    T[] nums; // array of Number or subclass
}

```

```

// Pass the constructor a reference to
// an array of type Number or subclass.
Stats(T[] o) {
    nums = o;
}

// Return type double in all cases.
double average() {
    double sum = 0.0;

    for(int i=0; i < nums.length; i++)
        sum += nums[i].doubleValue();

    return sum / nums.length;
}

// Demonstrate Stats.
class BoundsDemo {
    public static void main(String args[]) {

        Integer inums[] = { 1, 2, 3, 4, 5 };
        Stats<Integer> iob = new Stats<Integer>(inums);
        double v = iob.average();
        System.out.println("iob average is " + v);

        Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        Stats<Double> dob = new Stats<Double>(dnums);
        double w = dob.average();
        System.out.println("dob average is " + w);

        // This won't compile because String is not a
        // subclass of Number.
        // String strs[] = { "1", "2", "3", "4", "5" };
        // Stats<String> strob = new Stats<String>(strs);

        // double x = strob.average();
        // System.out.println("strob average is " + v);

    }
}

```

The output is shown here:

```

Average is 3.0
Average is 3.3

```

Notice how **Stats** is now declared by this line:

```
class Stats<T extends Number> {
```

Because the type **T** is now bounded by **Number**, the Java compiler knows that all objects of type **T** can call **doubleValue()** because it is a method declared by **Number**. This is, by itself, a major advantage. However, as an added bonus, the bounding of **T** also prevents nonnumeric **Stats** objects from being created. For example, if you try removing the comments from the lines at the end of the program, and then try recompiling, you will receive compile-time errors because **String** is not a subclass of **Number**.

In addition to using a class type as a bound, you can also use an interface type. In fact, you can specify multiple interfaces as bounds. Furthermore, a bound can include both a class type and one or more interfaces. In this case, the class type must be specified first. When a bound includes an interface type, only type arguments that implement that interface are legal. When specifying a bound that has a class and an interface, or multiple interfaces, use the **&** operator to connect them. For example,

```
class Gen<T extends MyClass & MyInterface> { // ...
```

Here, **T** is bounded by a class called **MyClass** and an interface called **MyInterface**. Thus, any type argument passed to **T** must be a subclass of **MyClass** and implement **MyInterface**.

Using Wildcard Arguments

As useful as type safety is, sometimes it can get in the way of perfectly acceptable constructs. For example, given the **Stats** class shown at the end of the preceding section, assume that you want to add a method called **sameAvg()** that determines if two **Stats** objects contain arrays that yield the same average, no matter what type of numeric data each object holds. For example, if one object contains the **double** values 1.0, 2.0, and 3.0, and the other object contains the integer values 2, 1, and 3, then the averages will be the same. One way to implement **sameAvg()** is to pass it a **Stats** argument, and then compare the average of that argument against the invoking object, returning true only if the averages are the same. For example, you want to be able to call **sameAvg()**, as shown here:

```
Integer inums[] = { 1, 2, 3, 4, 5 };
Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };

Stats<Integer> iob = new Stats<Integer>(inums);
Stats<Double> dob = new Stats<Double>(dnums);

if(iob.sameAvg(dob))
    System.out.println("Averages are the same.");
else
    System.out.println("Averages differ.");
```

At first, creating **sameAvg()** seems like an easy problem. Because **Stats** is generic and its **average()** method can work on any type of **Stats** object, it seems that creating **sameAvg()** would be straightforward. Unfortunately, trouble starts as soon as you try to declare a parameter of type **Stats**. Because **Stats** is a parameterized type, what do you specify for **Stats**' type parameter when you declare a parameter of that type?

At first, you might think of a solution like this, in which **T** is used as the type parameter:

```
// This won't work!
// Determine if two averages are the same.
boolean sameAvg(Stats<T> ob) {
    if(average() == ob.average())
        return true;

    return false;
}
```

The trouble with this attempt is that it will work only with other **Stats** objects whose type is the same as the invoking object. For example, if the invoking object is of type **Stats<Integer>**, then the parameter **ob** must also be of type **Stats<Integer>**. It can't be used to compare the average of an object of type **Stats<Double>** with the average of an object of type **Stats<Short>**, for example. Therefore, this approach won't work except in a very narrow context and does not yield a general (that is, generic) solution.

To create a generic **sameAvg()** method, you must use another feature of Java generics: the *wildcard* argument. The wildcard argument is specified by the ?, and it represents an unknown type. Using a wildcard, here is one way to write the **sameAvg()** method:

```
// Determine if two averages are the same.
// Notice the use of the wildcard.
boolean sameAvg(Stats<?> ob) {
    if(average() == ob.average())
        return true;

    return false;
}
```

Here, **Stats<?>** matches any **Stats** object, allowing any two **Stats** objects to have their averages compared. The following program demonstrates this:

```
// Use a wildcard.
class Stats<T extends Number> {
    T[] nums; // array of Number or subclass

    // Pass the constructor a reference to
    // an array of type Number or subclass.
    Stats(T[] o) {
        nums = o;
    }

    // Return type double in all cases.
    double average() {
        double sum = 0.0;

        for(int i=0; i < nums.length; i++)
            sum += nums[i].doubleValue();
```

```

        return sum / nums.length;
    }

    // Determine if two averages are the same.
    // Notice the use of the wildcard.
    boolean sameAvg(Stats<?> ob) {
        if(average() == ob.average())
            return true;

        return false;
    }
}

// Demonstrate wildcard.
class WildcardDemo {
    public static void main(String args[]) {
        Integer inums[] = { 1, 2, 3, 4, 5 };
        Stats<Integer> iob = new Stats<Integer>(inums);
        double v = iob.average();
        System.out.println("iob average is " + v);

        Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        Stats<Double> dob = new Stats<Double>(dnums);
        double w = dob.average();
        System.out.println("dob average is " + w);

        Float fnums[] = { 1.0F, 2.0F, 3.0F, 4.0F, 5.0F };
        Stats<Float> fob = new Stats<Float>(fnums);
        double x = fob.average();
        System.out.println("fob average is " + x);

        // See which arrays have same average.
        System.out.print("Averages of iob and dob ");
        if(iob.sameAvg(dob))
            System.out.println("are the same.");
        else
            System.out.println("differ.");

        System.out.print("Averages of iob and fob ");
        if(iob.sameAvg(fob))
            System.out.println("are the same.");
        else
            System.out.println("differ.");
    }
}

```

The output is shown here:

```

iob average is 3.0
dob average is 3.3
fob average is 3.0
Averages of iob and dob differ.
Averages of iob and fob are the same.

```

One last point: It is important to understand that the wildcard does not affect what type of **Stats** objects can be created. This is governed by the **extends** clause in the **Stats** declaration. The wildcard simply matches any *valid* **Stats** object.

Bounded Wildcards

Wildcard arguments can be bounded in much the same way that a type parameter can be bounded. A bounded wildcard is especially important when you are creating a generic type that will operate on a class hierarchy. To understand why, let's work through an example. Consider the following hierarchy of classes that encapsulate coordinates:

```
// Two-dimensional coordinates.
class TwoD {
    int x, y;

    TwoD(int a, int b) {
        x = a;
        y = b;
    }
}

// Three-dimensional coordinates.
class ThreeD extends TwoD {
    int z;

    ThreeD(int a, int b, int c) {
        super(a, b);
        z = c;
    }
}

// Four-dimensional coordinates.
class FourD extends ThreeD {
    int t;

    FourD(int a, int b, int c, int d) {
        super(a, b, c);
        t = d;
    }
}
```

At the top of the hierarchy is **TwoD**, which encapsulates a two-dimensional, XY coordinate. **TwoD** is inherited by **ThreeD**, which adds a third dimension, creating an XYZ coordinate. **ThreeD** is inherited by **FourD**, which adds a fourth dimension (time), yielding a four-dimensional coordinate.

Shown next is a generic class called **Coords**, which stores an array of coordinates:

```
// This class holds an array of coordinate objects.
class Coords<T extends TwoD> {
    T[] coords;

    Coords(T[] o) { coords = o; }
}
```

Notice that **Coords** specifies a type parameter bounded by **TwoD**. This means that any array stored in a **Coords** object will contain objects of type **TwoD** or one of its subclasses.

Now, assume that you want to write a method that displays the X and Y coordinates for each element in the **coords** array of a **Coords** object. Because all types of **Coords** objects have at least two coordinates (X and Y), this is easy to do using a wildcard, as shown here:

```
static void showXY(Coords<?> c) {
    System.out.println("X Y Coordinates:");
    for(int i=0; i < c.coords.length; i++)
        System.out.println(c.coords[i].x + " " +
                           c.coords[i].y);
    System.out.println();
}
```

Because **Coords** is a bounded generic type that specifies **TwoD** as an upper bound, all objects that can be used to create a **Coords** object will be arrays of type **TwoD**, or of classes derived from **TwoD**. Thus, **showXY()** can display the contents of any **Coords** object.

However, what if you want to create a method that displays the X, Y, and Z coordinates of a **ThreeD** or **FourD** object? The trouble is that not all **Coords** objects will have three coordinates, because a **Coords<TwoD>** object will only have X and Y. Therefore, how do you write a method that displays the X, Y, and Z coordinates for **Coords<ThreeD>** and **Coords<FourD>** objects, while preventing that method from being used with **Coords<TwoD>** objects? The answer is the *bounded wildcard argument*.

A bounded wildcard specifies either an upper bound or a lower bound for the type argument. This enables you to restrict the types of objects upon which a method will operate. The most common bounded wildcard is the upper bound, which is created using an **extends** clause in much the same way it is used to create a bounded type.

Using a bounded wildcard, it is easy to create a method that displays the X, Y, and Z coordinates of a **Coords** object, if that object actually has those three coordinates. For example, the following **showXYZ()** method shows the X, Y, and Z coordinates of the elements stored in a **Coords** object, if those elements are actually of type **ThreeD** (or are derived from **ThreeD**):

```
static void showXYZ(Coords<? extends ThreeD> c) {
    System.out.println("X Y Z Coordinates:");
    for(int i=0; i < c.coords.length; i++)
        System.out.println(c.coords[i].x + " " +
                           c.coords[i].y + " " +
                           c.coords[i].z);
    System.out.println();
}
```

Notice that an **extends** clause has been added to the wildcard in the declaration of parameter **c**. It states that the **?** can match any type as long as it is **ThreeD**, or a class derived from **ThreeD**. Thus, the **extends** clause establishes an upper bound that the **?** can match. Because of this bound, **showXYZ()** can be called with references to objects of type **Coords<ThreeD>** or **Coords<FourD>**, but not with a reference of type **Coords<TwoD>**. Attempting to call **showXYZ()** with a **Coords<TwoD>** reference results in a compile-time error, thus ensuring type safety.

Here is an entire program that demonstrates the actions of a bounded wildcard argument:

```
// Bounded Wildcard arguments.

// Two-dimensional coordinates.
class TwoD {
    int x, y;

    TwoD(int a, int b) {
        x = a;
        y = b;
    }
}

// Three-dimensional coordinates.
class ThreeD extends TwoD {
    int z;

    ThreeD(int a, int b, int c) {
        super(a, b);
        z = c;
    }
}

// Four-dimensional coordinates.
class FourD extends ThreeD {
    int t;

    FourD(int a, int b, int c, int d) {
        super(a, b, c);
        t = d;
    }
}

// This class holds an array of coordinate objects.
class Coords<T extends TwoD> {
    T[] coords;

    Coords(T[] o) { coords = o; }
}

// Demonstrate a bounded wildcard.
class BoundedWildcard {
    static void showXY(Coords<?> c) {
        System.out.println("X Y Coordinates:");
        for(int i=0; i < c.coords.length; i++)
            System.out.println(c.coords[i].x + " " +
                               c.coords[i].y);
        System.out.println();
    }

    static void showXYZ(Coords<? extends ThreeD> c) {
        System.out.println("X Y Z Coordinates:");
        for(int i=0; i < c.coords.length; i++)
    }
```

```

        System.out.println(c.coords[i].x + " " +
                           c.coords[i].y + " " +
                           c.coords[i].z);
        System.out.println();
    }

    static void showAll(Coords<? extends FourD> c) {
        System.out.println("X Y Z T Coordinates:");
        for(int i=0; i < c.coords.length; i++)
            System.out.println(c.coords[i].x + " " +
                               c.coords[i].y + " " +
                               c.coords[i].z + " " +
                               c.coords[i].t);
        System.out.println();
    }

    public static void main(String args[]) {
        TwoD td[] = {
            new TwoD(0, 0),
            new TwoD(7, 9),
            new TwoD(18, 4),
            new TwoD(-1, -23)
        };
        Coords<TwoD> tdlocs = new Coords<TwoD>(td);

        System.out.println("Contents of tdlocs.");
        showXY(tdlocs); // OK, is a TwoD
//        showXYZ(tdlocs); // Error, not a ThreeD
//        showAll(tdlocs); // Error, not a FourD

        // Now, create some FourD objects.
        FourD fd[] = {
            new FourD(1, 2, 3, 4),
            new FourD(6, 8, 14, 8),
            new FourD(22, 9, 4, 9),
            new FourD(3, -2, -23, 17)
        };
        Coords<FourD> fdlocs = new Coords<FourD>(fd);

        System.out.println("Contents of fdlocs.");
        // These are all OK.
        showXY(fdlocs);
        showXYZ(fdlocs);
        showAll(fdlocs);
    }
}

```

The output from the program is shown here:

```

Contents of tdlocs.
X Y Coordinates:
0 0

```

```

7 9
18 4
-1 -23

Contents of fdlocs.
X Y Coordinates:
1 2
6 8
22 9
3 -2

X Y Z Coordinates:
1 2 3
6 8 14
22 9 4
3 -2 -23

```

```

X Y Z T Coordinates:
1 2 3 4
6 8 14 8
22 9 4 9
3 -2 -23 17

```

Notice these commented-out lines:

```
// showXYZ(tdlocs); // Error, not a ThreeD
// showAll(tdlocs); // Error, not a FourD
```

Because **tdlocs** is a **Coords(TwoD)** object, it cannot be used to call **showXYZ()** or **showAll()** because bounded wildcard arguments in their declarations prevent it. To prove this to yourself, try removing the comment symbols, and then attempt to compile the program. You will receive compilation errors because of the type mismatches.

In general, to establish an upper bound for a wildcard, use the following type of wildcard expression:

`<? extends superclass>`

where *superclass* is the name of the class that serves as the upper bound. Remember, this is an inclusive clause because the class forming the upper bound (that is, specified by *superclass*) is also within bounds.

You can also specify a lower bound for a wildcard by adding a **super** clause to a wildcard declaration. Here is its general form:

`<? super subclass>`

In this case, only classes that are superclasses of *subclass* are acceptable arguments. This is an inclusive clause.

Creating a Generic Method

As the preceding examples have shown, methods inside a generic class can make use of a class' type parameter and are, therefore, automatically generic relative to the type parameter. However, it is possible to declare a generic method that uses one or more type parameters of

its own. Furthermore, it is possible to create a generic method that is enclosed within a non-generic class.

Let's begin with an example. The following program declares a non-generic class called **GenMethDemo** and a static generic method within that class called **isIn()**. The **isIn()** method determines if an object is a member of an array. It can be used with any type of object and array as long as the array contains objects that are compatible with the type of the object being sought.

```
// Demonstrate a simple generic method.
class GenMethDemo {

    // Determine if an object is in an array.
    static <T extends Comparable<T>, V extends T> boolean isIn(T x, V[] y) {
        for(int i=0; i < y.length; i++)
            if(x.equals(y[i])) return true;

        return false;
    }

    public static void main(String args[]) {
        // Use isIn() on Integers.
        Integer nums[] = { 1, 2, 3, 4, 5 };

        if(isIn(2, nums))
            System.out.println("2 is in nums");

        if(!isIn(7, nums))
            System.out.println("7 is not in nums");

        System.out.println();

        // Use isIn() on Strings.
        String strs[] = { "one", "two", "three",
                          "four", "five" };

        if(isIn("two", strs))
            System.out.println("two is in strs");

        if(!isIn("seven", strs))
            System.out.println("seven is not in strs");

        // Oops! Won't compile! Types must be compatible.
        //     if(isIn("two", nums))
        //         System.out.println("two is in strs");
    }
}
```

The output from the program is shown here:

```
2 is in nums
7 is not in nums
```

```
two is in strs
seven is not in strs
```

Let's examine **isIn()** closely. First, notice how it is declared by this line:

```
static <T extends Comparable<T>, V extends T> boolean isIn(T x, V[] y) {
```

The type parameters are declared *before* the return type of the method. Also note that **T** extends **Comparable<T>**. **Comparable** is an interface declared in **java.lang**. A class that implements **Comparable** defines objects that can be ordered. Thus, requiring an upper bound of **Comparable** ensures that **isIn()** can be used only with objects that are capable of being compared. **Comparable** is generic, and its type parameter specifies the type of objects that it compares. (Shortly, you will see how to create a generic interface.) Next, notice that the type **V** is upper-bounded by **T**. Thus, **V** must either be the same as type **T**, or a subclass of **T**. This relationship enforces that **isIn()** can be called only with arguments that are compatible with each other. Also notice that **isIn()** is static, enabling it to be called independently of any object. Understand, though, that generic methods can be either static or non-static. There is no restriction in this regard.

Now, notice how **isIn()** is called within **main()** by use of the normal call syntax, without the need to specify type arguments. This is because the types of the arguments are automatically discerned, and the types of **T** and **V** are adjusted accordingly. For example, in the first call:

```
if(isIn(2, nums))
```

the type of the first argument is **Integer** (due to autoboxing), which causes **Integer** to be substituted for **T**. The base type of the second argument is also **Integer**, which makes **Integer** a substitute for **V**, too. In the second call, **String** types are used, and the types of **T** and **V** are replaced by **String**.

Although type inference will be sufficient for most generic method calls, you can explicitly specify the type argument if needed. For example, here is how the first call to **isIn()** looks when the type arguments are specified:

```
GenMethDemo.<Integer, Integer>isIn(2, nums)
```

Of course, in this case, there is nothing gained by specifying the type arguments. Furthermore, JDK 8 has improved type inference as it relates to methods. As a result, there are fewer cases in which explicit type arguments are needed.

Now, notice the commented-out code, shown here:

```
//     if(isIn("two", nums))
//         System.out.println("two is in strs");
```

If you remove the comments and then try to compile the program, you will receive an error. The reason is that the type parameter **V** is bounded by **T** in the **extends** clause in **V**'s declaration. This means that **V** must be either type **T**, or a subclass of **T**. In this case, the first argument is of type **String**, making **T** into **String**, but the second argument is of type

Integer, which is not a subclass of **String**. This causes a compile-time type-mismatch error. This ability to enforce type safety is one of the most important advantages of generic methods.

The syntax used to create **isIn()** can be generalized. Here is the syntax for a generic method:

```
<type-param-list> ret-type meth-name (param-list) { // ...
```

In all cases, *type-param-list* is a comma-separated list of type parameters. Notice that for a generic method, the type parameter list precedes the return type.

Generic Constructors

It is possible for constructors to be generic, even if their class is not. For example, consider the following short program:

```
// Use a generic constructor.
class GenCons {
    private double val;

    <T extends Number> GenCons(T arg) {
        val = arg.doubleValue();
    }

    void showval() {
        System.out.println("val: " + val);
    }
}

class GenConsDemo {
    public static void main(String args[]) {
        GenCons test = new GenCons(100);
        GenCons test2 = new GenCons(123.5F);

        test.showval();
        test2.showval();
    }
}
```

The output is shown here:

```
val: 100.0
val: 123.5
```

Because **GenCons()** specifies a parameter of a generic type, which must be a subclass of **Number**, **GenCons()** can be called with any numeric type, including **Integer**, **Float**, or **Double**. Therefore, even though **GenCons** is not a generic class, its constructor is generic.

Generic Interfaces

In addition to generic classes and methods, you can also have generic interfaces. Generic interfaces are specified just like generic classes. Here is an example. It creates an interface called **MinMax** that declares the methods **min()** and **max()**, which are expected to return the minimum and maximum value of some set of objects.

```
// A generic interface example.

// A Min/Max interface.
interface MinMax<T extends Comparable<T>> {
    T min();
    T max();
}

// Now, implement MinMax
class MyClass<T extends Comparable<T>> implements MinMax<T> {
    T[] vals;

    MyClass(T[] o) { vals = o; }

    // Return the minimum value in vals.
    public T min() {
        T v = vals[0];

        for(int i=1; i < vals.length; i++)
            if(vals[i].compareTo(v) < 0) v = vals[i];

        return v;
    }

    // Return the maximum value in vals.
    public T max() {
        T v = vals[0];

        for(int i=1; i < vals.length; i++)
            if(vals[i].compareTo(v) > 0) v = vals[i];

        return v;
    }
}

class GenIFDemo {
    public static void main(String args[]) {
        Integer inums[] = {3, 6, 2, 8, 6 };
        Character chs[] = {'b', 'r', 'p', 'w' };

        MyClass<Integer> iob = new MyClass<Integer>(inums);
        MyClass<Character> cob = new MyClass<Character>(chs);

        System.out.println("Max value in inums: " + iob.max());
        System.out.println("Min value in inums: " + iob.min());
```

```

        System.out.println("Max value in chs: " + cob.max());
        System.out.println("Min value in chs: " + cob.min());
    }
}

```

The output is shown here:

```

Max value in inums: 8
Min value in inums: 2
Max value in chs: w
Min value in chs: b

```

Although most aspects of this program should be easy to understand, a couple of key points need to be made. First, notice that **MinMax** is declared like this:

```
interface MinMax<T extends Comparable<T>> {
```

In general, a generic interface is declared in the same way as is a generic class. In this case, the type parameter is **T**, and its upper bound is **Comparable**. As explained earlier, **Comparable** is an interface defined by **java.lang** that specifies how objects are compared. Its type parameter specifies the type of the objects being compared.

Next, **MinMax** is implemented by **MyClass**. Notice the declaration of **MyClass**, shown here:

```
class MyClass<T extends Comparable<T>> implements MinMax<T> {
```

Pay special attention to the way that the type parameter **T** is declared by **MyClass** and then passed to **MinMax**. Because **MinMax** requires a type that implements **Comparable**, the implementing class (**MyClass** in this case) must specify the same bound. Furthermore, once this bound has been established, there is no need to specify it again in the **implements** clause. In fact, it would be wrong to do so. For example, this line is incorrect and won't compile:

```
// This is wrong!
class MyClass<T extends Comparable<T>>
    implements MinMax<T extends Comparable<T>> {
```

Once the type parameter has been established, it is simply passed to the interface without further modification.

In general, if a class implements a generic interface, then that class must also be generic, at least to the extent that it takes a type parameter that is passed to the interface. For example, the following attempt to declare **MyClass** is in error:

```
class MyClass implements MinMax<T> { // Wrong!
```

Because **MyClass** does not declare a type parameter, there is no way to pass one to **MinMax**. In this case, the identifier **T** is simply unknown, and the compiler reports an error. Of course, if a class implements a *specific type* of generic interface, such as shown here:

```
class MyClass implements MinMax<Integer> { // OK
```

then the implementing class does not need to be generic.

The generic interface offers two benefits. First, it can be implemented for different types of data. Second, it allows you to put constraints (that is, bounds) on the types of data for which the interface can be implemented. In the **MinMax** example, only types that implement the **Comparable** interface can be passed to **T**.

Here is the generalized syntax for a generic interface:

```
interface interface-name<type-param-list> { // ...}
```

Here, *type-param-list* is a comma-separated list of type parameters. When a generic interface is implemented, you must specify the type arguments, as shown here:

```
class class-name<type-param-list>
    implements interface-name<type-arg-list> {
```

Raw Types and Legacy Code

Because support for generics did not exist prior to JDK 5, it was necessary to provide some transition path from old, pre-generics code. At the time of this writing, there is still pre-generics legacy code that must remain both functional and compatible with generics. Pre-generics code must be able to work with generics, and generic code must be able to work with pre-generics code.

To handle the transition to generics, Java allows a generic class to be used without any type arguments. This creates a *raw type* for the class. This raw type is compatible with legacy code, which has no knowledge of generics. The main drawback to using the raw type is that the type safety of generics is lost.

Here is an example that shows a raw type in action:

```
// Demonstrate a raw type.
class Gen<T> {

    T ob; // declare an object of type T

    // Pass the constructor a reference to
    // an object of type T.
    Gen(T o) {
        ob = o;
    }

    // Return ob.
    T getob() {
        return ob;
    }
}

// Demonstrate raw type.
class RawDemo {
    public static void main(String args[]) {

        // Create a Gen object for Integers.
        Gen<Integer> iOb = new Gen<Integer>(88);
```

```

// Create a Gen object for Strings.
Gen<String> strOb = new Gen<String>("Generics Test");

// Create a raw-type Gen object and give it
// a Double value.
Gen raw = new Gen(new Double(98.6));

// Cast here is necessary because type is unknown.
double d = (Double) raw.getob();
System.out.println("value: " + d);

// The use of a raw type can lead to run-time
// exceptions. Here are some examples.

// The following cast causes a run-time error!
// int i = (Integer) raw.getob(); // run-time error

// This assignment overrides type safety.
strOb = raw; // OK, but potentially wrong
// String str = strOb.getob(); // run-time error

// This assignment also overrides type safety.
raw = iOb; // OK, but potentially wrong
// d = (Double) raw.getob(); // run-time error
}
}

```

This program contains several interesting things. First, a raw type of the generic **Gen** class is created by the following declaration:

```
Gen raw = new Gen(new Double(98.6));
```

Notice that no type arguments are specified. In essence, this creates a **Gen** object whose type **T** is replaced by **Object**.

A raw type is not type safe. Thus, a variable of a raw type can be assigned a reference to any type of **Gen** object. The reverse is also allowed; a variable of a specific **Gen** type can be assigned a reference to a raw **Gen** object. However, both operations are potentially unsafe because the type checking mechanism of generics is circumvented.

This lack of type safety is illustrated by the commented-out lines at the end of the program. Let's examine each case. First, consider the following situation:

```
// int i = (Integer) raw.getob(); // run-time error
```

In this statement, the value of **ob** inside **raw** is obtained, and this value is cast to **Integer**. The trouble is that **raw** contains a **Double** value, not an integer value. However, this cannot be detected at compile time because the type of **raw** is unknown. Thus, this statement fails at run time.

The next sequence assigns to a **strOb** (a reference of type **Gen<String>**) a reference to a raw **Gen** object:

```
strOb = raw; // OK, but potentially wrong
// String str = strOb.getob(); // run-time error
```

The assignment, itself, is syntactically correct, but questionable. Because `strOb` is of type `Gen<String>`, it is assumed to contain a `String`. However, after the assignment, the object referred to by `strOb` contains a `Double`. Thus, at run time, when an attempt is made to assign the contents of `strOb` to `str`, a run-time error results because `strOb` now contains a `Double`. Thus, the assignment of a raw reference to a generic reference bypasses the type-safety mechanism.

The following sequence inverts the preceding case:

```
raw = iOb; // OK, but potentially wrong
// d = (Double) raw.getob(); // run-time error
```

Here, a generic reference is assigned to a raw reference variable. Although this is syntactically correct, it can lead to problems, as illustrated by the second line. In this case, `raw` now refers to an object that contains an `Integer` object, but the cast assumes that it contains a `Double`. This error cannot be prevented at compile time. Rather, it causes a run-time error.

Because of the potential for danger inherent in raw types, `javac` displays *unchecked warnings* when a raw type is used in a way that might jeopardize type safety. In the preceding program, these lines generate unchecked warnings:

```
Gen raw = new Gen(new Double(98.6));
strOb = raw; // OK, but potentially wrong
```

In the first line, it is the call to the `Gen` constructor without a type argument that causes the warning. In the second line, it is the assignment of a raw reference to a generic variable that generates the warning.

At first, you might think that this line should also generate an unchecked warning, but it does not:

```
raw = iOb; // OK, but potentially wrong
```

No compiler warning is issued because the assignment does not cause any *further* loss of type safety than had already occurred when `raw` was created.

One final point: You should limit the use of raw types to those cases in which you must mix legacy code with newer, generic code. Raw types are simply a transitional feature and not something that should be used for new code.

Generic Class Hierarchies

Generic classes can be part of a class hierarchy in just the same way as a non-generic class. Thus, a generic class can act as a superclass or be a subclass. The key difference between generic and non-generic hierarchies is that in a generic hierarchy, any type arguments needed by a generic superclass must be passed up the hierarchy by all subclasses. This is similar to the way that constructor arguments must be passed up a hierarchy.

Using a Generic Superclass

Here is a simple example of a hierarchy that uses a generic superclass:

```
// A simple generic class hierarchy.
class Gen<T> {
    T ob;

    Gen(T o) {
        ob = o;
    }

    // Return ob.
    T getob() {
        return ob;
    }
}

// A subclass of Gen.
class Gen2<T> extends Gen<T> {
    Gen2(T o) {
        super(o);
    }
}
```

In this hierarchy, **Gen2** extends the generic class **Gen**. Notice how **Gen2** is declared by the following line:

```
class Gen2<T> extends Gen<T> {
```

The type parameter **T** is specified by **Gen2** and is also passed to **Gen** in the **extends** clause. This means that whatever type is passed to **Gen2** will also be passed to **Gen**. For example, this declaration,

```
Gen2<Integer> num = new Gen2<Integer>(100);
```

passes **Integer** as the type parameter to **Gen**. Thus, the **ob** inside the **Gen** portion of **Gen2** will be of type **Integer**.

Notice also that **Gen2** does not use the type parameter **T** except to support the **Gen** superclass. Thus, even if a subclass of a generic superclass would otherwise not need to be generic, it still must specify the type parameter(s) required by its generic superclass.

Of course, a subclass is free to add its own type parameters, if needed. For example, here is a variation on the preceding hierarchy in which **Gen2** adds a type parameter of its own:

```
// A subclass can add its own type parameters.
class Gen<T> {
    T ob; // declare an object of type T

    // Pass the constructor a reference to
    // an object of type T.
```

```

    Gen(T o) {
        ob = o;
    }

    // Return ob.
    T getob() {
        return ob;
    }
}

// A subclass of Gen that defines a second
// type parameter, called V.
class Gen2<T, V> extends Gen<T> {
    V ob2;

    Gen2(T o, V o2) {
        super(o);
        ob2 = o2;
    }

    V getob2() {
        return ob2;
    }
}

// Create an object of type Gen2.
class HierDemo {
    public static void main(String args[]) {

        // Create a Gen2 object for String and Integer.
        Gen2<String, Integer> x =
            new Gen2<String, Integer>("Value is: ", 99);

        System.out.print(x.getob());
        System.out.println(x.getob2());
    }
}

```

Notice the declaration of this version of **Gen2**, which is shown here:

```
class Gen2<T, V> extends Gen<T> {
```

Here, **T** is the type passed to **Gen**, and **V** is the type that is specific to **Gen2**. **V** is used to declare an object called **ob2**, and as a return type for the method **getob2()**. In **main()**, a **Gen2** object is created in which type parameter **T** is **String**, and type parameter **V** is **Integer**. The program displays the following, expected, result:

```
Value is: 99
```

A Generic Subclass

It is perfectly acceptable for a non-generic class to be the superclass of a generic subclass. For example, consider this program:

```
// A non-generic class can be the superclass
// of a generic subclass.

// A non-generic class.
class NonGen {
    int num;

    NonGen(int i) {
        num = i;
    }

    int getnum() {
        return num;
    }
}

// A generic subclass.
class Gen<T> extends NonGen {
    T ob; // declare an object of type T

    // Pass the constructor a reference to
    // an object of type T.
    Gen(T o, int i) {
        super(i);
        ob = o;
    }

    // Return ob.
    T getob() {
        return ob;
    }
}

// Create a Gen object.
class HierDemo2 {
    public static void main(String args[]) {

        // Create a Gen object for String.
        Gen<String> w = new Gen<String>("Hello", 47);

        System.out.print(w.getob() + " ");
        System.out.println(w.getnum());
    }
}
```

The output from the program is shown here:

```
Hello 47
```

In the program, notice how **Gen** inherits **NonGen** in the following declaration:

```
class Gen<T> extends NonGen {
```

Because **NonGen** is not generic, no type argument is specified. Thus, even though **Gen** declares the type parameter **T**, it is not needed by (nor can it be used by) **NonGen**. Thus, **NonGen** is inherited by **Gen** in the normal way. No special conditions apply.

Run-Time Type Comparisons Within a Generic Hierarchy

Recall the run-time type information operator **instanceof** that was described in Chapter 13. As explained, **instanceof** determines if an object is an instance of a class. It returns true if an object is of the specified type or can be cast to the specified type. The **instanceof** operator can be applied to objects of generic classes. The following class demonstrates some of the type compatibility implications of a generic hierarchy:

```
// Use the instanceof operator with a generic class hierarchy.
class Gen<T> {
    T ob;

    Gen(T o) {
        ob = o;
    }

    // Return ob.
    T getob() {
        return ob;
    }
}

// A subclass of Gen.
class Gen2<T> extends Gen<T> {
    Gen2(T o) {
        super(o);
    }
}

// Demonstrate run-time type ID implications of generic
// class hierarchy.
class HierDemo3 {
    public static void main(String args[]) {

        // Create a Gen object for Integers.
        Gen<Integer> iOb = new Gen<Integer>(88);

        // Create a Gen2 object for Integers.
        Gen2<Integer> iOb2 = new Gen2<Integer>(99);
```

```

// Create a Gen2 object for Strings.
Gen2<String> strOb2 = new Gen2<String>("Generics Test");

// See if iOb2 is some form of Gen2.
if(iOb2 instanceof Gen2<?>)
    System.out.println("iOb2 is instance of Gen2");

// See if iOb2 is some form of Gen.
if(iOb2 instanceof Gen<?>)
    System.out.println("iOb2 is instance of Gen");

System.out.println();

// See if strOb2 is a Gen2.
if(strOb2 instanceof Gen2<?>)
    System.out.println("strOb2 is instance of Gen2");

// See if strOb2 is a Gen.
if(strOb2 instanceof Gen<?>)
    System.out.println("strOb2 is instance of Gen");

System.out.println();

// See if iOb is an instance of Gen2, which it is not.
if(iOb instanceof Gen2<?>)
    System.out.println("iOb is instance of Gen2");

// See if iOb is an instance of Gen, which it is.
if(iOb instanceof Gen<?>)
    System.out.println("iOb is instance of Gen");

// The following can't be compiled because
// generic type info does not exist at run time.
// if(iOb2 instanceof Gen2<Integer>)
//     System.out.println("iOb2 is instance of Gen2<Integer>");
}
}

```

The output from the program is shown here:

```

iOb2 is instance of Gen2
iOb2 is instance of Gen

strOb2 is instance of Gen2
strOb2 is instance of Gen

iOb is instance of Gen

```

In this program, **Gen2** is a subclass of **Gen**, which is generic on type parameter **T**. In **main()**, three objects are created. The first is **iOb**, which is an object of type **Gen<Integer>**. The second is **iOb2**, which is an instance of **Gen2<Integer>**. Finally, **strOb2** is an object of type **Gen2<String>**.

Then, the program performs these **instanceof** tests on the type of **iOb2**:

```
// See if iOb2 is some form of Gen2.
if(iOb2 instanceof Gen2<?>)
    System.out.println("iOb2 is instance of Gen2");

// See if iOb2 is some form of Gen.
if(iOb2 instanceof Gen<?>)
    System.out.println("iOb2 is instance of Gen");
```

As the output shows, both succeed. In the first test, **iOb2** is checked against **Gen2<?>**. This test succeeds because it simply confirms that **iOb2** is an object of some type of **Gen2** object. The use of the wildcard enables **instanceof** to determine if **iOb2** is an object of any type of **Gen2**. Next, **iOb2** is tested against **Gen<?>**, the superclass type. This is also true because **iOb2** is some form of **Gen**, the superclass. The next few lines in **main()** show the same sequence (and same results) for **strOb2**.

Next, **iOb**, which is an instance of **Gen<Integer>** (the superclass), is tested by these lines:

```
// See if iOb is an instance of Gen2, which it is not.
if(iOb instanceof Gen2<?>)
    System.out.println("iOb is instance of Gen2");

// See if iOb is an instance of Gen, which it is.
if(iOb instanceof Gen<?>)
    System.out.println("iOb is instance of Gen");
```

The first **if** fails because **iOb** is not some type of **Gen2** object. The second test succeeds because **iOb** is some type of **Gen** object.

Now, look closely at these commented-out lines:

```
// The following can't be compiled because
// generic type info does not exist at run time.
//     if(iOb2 instanceof Gen2<Integer>)
//         System.out.println("iOb2 is instance of Gen2<Integer>");
```

As the comments indicate, these lines can't be compiled because they attempt to compare **iOb2** with a specific type of **Gen2**, in this case, **Gen2<Integer>**. Remember, there is no generic type information available at run time. Therefore, there is no way for **instanceof** to know if **iOb2** is an instance of **Gen2<Integer>** or not.

Casting

You can cast one instance of a generic class into another only if the two are otherwise compatible and their type arguments are the same. For example, assuming the foregoing program, this cast is legal:

```
(Gen<Integer>) iOb2 // legal
```

because **iOb2** includes an instance of **Gen<Integer>**. But, this cast:

```
(Gen<Long>) iOb2 // illegal
```

is not legal because **iOb2** is not an instance of **Gen<Long>**.

Overriding Methods in a Generic Class

A method in a generic class can be overridden just like any other method. For example, consider this program in which the method **getob()** is overridden:

```
// Overriding a generic method in a generic class.
class Gen<T> {
    T ob; // declare an object of type T

    // Pass the constructor a reference to
    // an object of type T.
    Gen(T o) {
        ob = o;
    }

    // Return ob.
    T getob() {
        System.out.print("Gen's getob(): " );
        return ob;
    }
}

// A subclass of Gen that overrides getob().
class Gen2<T> extends Gen<T> {

    Gen2(T o) {
        super(o);
    }

    // Override getob().
    T getob() {
        System.out.print("Gen2's getob(): " );
        return ob;
    }
}

// Demonstrate generic method override.
class OverrideDemo {
    public static void main(String args[]) {

        // Create a Gen object for Integers.
        Gen<Integer> iOb = new Gen<Integer>(88);

        // Create a Gen2 object for Integers.
        Gen2<Integer> iOb2 = new Gen2<Integer>(99);

        // Create a Gen2 object for Strings.
        Gen2<String> strOb2 = new Gen2<String> ("Generics Test");

        System.out.println(iOb.getob());
        System.out.println(iOb2.getob());
        System.out.println(strOb2.getob());
    }
}
```

The output is shown here:

```
Gen's getob(): 88
Gen2's getob(): 99
Gen2's getob(): Generics Test
```

As the output confirms, the overridden version of **getob()** is called for objects of type **Gen2**, but the superclass version is called for objects of type **Gen**.

Type Inference with Generics

Beginning with JDK 7, it is possible to shorten the syntax used to create an instance of a generic type. To begin, consider the following generic class:

```
class MyClass<T, V> {
    T ob1;
    V ob2;

    MyClass(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }
    // ...
}
```

Prior to JDK 7, to create an instance of **MyClass**, you would have needed to use a statement similar to the following:

```
MyClass<Integer, String> mcOb =
    new MyClass<Integer, String>(98, "A String");
```

Here, the type arguments (which are **Integer** and **String**) are specified twice: first, when **mcOb** is declared, and second, when a **MyClass** instance is created via **new**. Since generics were introduced by JDK 5, this is the form required by all versions of Java prior to JDK 7. Although there is nothing wrong, per se, with this form, it is a bit more verbose than it needs to be. In the **new** clause, the type of the type arguments can be readily inferred from the type of **mcOb**; therefore, there is really no reason that they need to be specified a second time. To address this situation, JDK 7 added a syntactic element that lets you avoid the second specification.

Today the preceding declaration can be rewritten as shown here:

```
MyClass<Integer, String> mcOb = new MyClass<>(98, "A String");
```

Notice that the instance creation portion simply uses **<>**, which is an empty type argument list. This is referred to as the *diamond* operator. It tells the compiler to infer the type arguments needed by the constructor in the **new** expression. The principal advantage of this type-inference syntax is that it shortens what are sometimes quite long declaration statements.

The preceding can be generalized. When type inference is used, the declaration syntax for a generic reference and instance creation has this general form:

```
class-name<type-arg-list> var-name = new class-name<>(cons-arg-list);
```

Here, the type argument list of the constructor in the `new` clause is empty.

Type inference can also be applied to parameter passing. For example, if the following method is added to `MyClass`,

```
boolean isSame(MyClass<T, V> o) {  
    if(ob1 == o.ob1 && ob2 == o.ob2) return true;  
    else return false;  
}
```

then the following call is legal:

```
if(mcOb.isSame(new MyClass<>(1, "test"))) System.out.println("Same");
```

In this case, the type arguments for the argument passed to `isSame()` can be inferred from the parameter's type.

Because the type-inference syntax was added by JDK 7 and won't work with older compilers, most of the examples in this book will continue to use the full syntax when declaring instances of generic classes. This way, the examples will work with any Java compiler that supports generics. Using the full-length syntax also makes it very clear precisely what is being created, which is important in example code shown in a book. However, in your own code, the use of the type-inference syntax will streamline your declarations.

ErasurE

Usually, it is not necessary to know the details about how the Java compiler transforms your source code into object code. However, in the case of generics, some general understanding of the process is important because it explains why the generic features work as they do—and why their behavior is sometimes a bit surprising. For this reason, a brief discussion of how generics are implemented in Java is in order.

An important constraint that governed the way that generics were added to Java was the need for compatibility with previous versions of Java. Simply put, generic code had to be compatible with preexisting, non-generic code. Thus, any changes to the syntax of the Java language, or to the JVM, had to avoid breaking older code. The way Java implements generics while satisfying this constraint is through the use of *erasure*.

In general, here is how erasure works. When your Java code is compiled, all generic type information is removed (erased). This means replacing type parameters with their bound type, which is `Object` if no explicit bound is specified, and then applying the appropriate casts (as determined by the type arguments) to maintain type compatibility with the types specified by the type arguments. The compiler also enforces this type compatibility. This approach to generics means that no type parameters exist at run time. They are simply a source-code mechanism.

Bridge Methods

Occasionally, the compiler will need to add a *bridge method* to a class to handle situations in which the type erasure of an overriding method in a subclass does not produce the same erasure as the method in the superclass. In this case, a method is generated that uses the type erasure of the superclass, and this method calls the method that has the type erasure specified by the subclass. Of course, bridge methods only occur at the bytecode level, are not seen by you, and are not available for your use.

Although bridge methods are not something that you will normally need to be concerned with, it is still instructive to see a situation in which one is generated. Consider the following program:

```
// A situation that creates a bridge method.
class Gen<T> {
    T ob; // declare an object of type T

    // Pass the constructor a reference to
    // an object of type T.
    Gen(T o) {
        ob = o;
    }

    // Return ob.
    T getob() {
        return ob;
    }
}

// A subclass of Gen.
class Gen2 extends Gen<String> {

    Gen2(String o) {
        super(o);
    }

    // A String-specific override of getob().
    String getob() {
        System.out.print("You called String getob(): ");
        return ob;
    }
}

// Demonstrate a situation that requires a bridge method.
class BridgeDemo {
    public static void main(String args[]) {

        // Create a Gen2 object for Strings.
        Gen2 strOb2 = new Gen2("Generics Test");

        System.out.println(strOb2.getob());
    }
}
```

In the program, the subclass **Gen2** extends **Gen**, but does so using a **String**-specific version of **Gen**, as its declaration shows:

```
class Gen2 extends Gen<String> {
```

Furthermore, inside **Gen2**, **getob()** is overridden with **String** specified as the return type:

```
// A String-specific override of getob().
String getob() {
    System.out.print("You called String getob(): ");
    return ob;
}
```

All of this is perfectly acceptable. The only trouble is that because of type erasure, the expected form of **getob()** will be

```
Object getob() { // ...
```

To handle this problem, the compiler generates a bridge method with the preceding signature that calls the **String** version. Thus, if you examine the class file for **Gen2** by using **javap**, you will see the following methods:

```
class Gen2 extends Gen<java.lang.String> {
    Gen2(java.lang.String);
    java.lang.String getob();
    java.lang.Object getob(); // bridge method
}
```

As you can see, the bridge method has been included. (The comment was added by the author and not by **javap**, and the precise output you see may vary based on the version of Java that you are using.)

There is one last point to make about this example. Notice that the only difference between the two **getob()** methods is their return type. Normally, this would cause an error, but because this does not occur in your source code, it does not cause a problem and is handled correctly by the JVM.

Ambiguity Errors

The inclusion of generics gives rise to a new type of error that you must guard against: *ambiguity*. Ambiguity errors occur when erasure causes two seemingly distinct generic declarations to resolve to the same erased type, causing a conflict. Here is an example that involves method overloading:

```
// Ambiguity caused by erasure on
// overloaded methods.
class MyGenClass<T, V> {
    T ob1;
    V ob2;
    // ...
```

```
// These two overloaded methods are ambiguous
// and will not compile.
void set(T o) {
    ob1 = o;
}

void set(V o) {
    ob2 = o;
}
}
```

Notice that **MyGenClass** declares two generic types: **T** and **V**. Inside **MyGenClass**, an attempt is made to overload **set()** based on parameters of type **T** and **V**. This looks reasonable because **T** and **V** appear to be different types. However, there are two ambiguity problems here.

First, as **MyGenClass** is written, there is no requirement that **T** and **V** actually be different types. For example, it is perfectly correct (in principle) to construct a **MyGenClass** object as shown here:

```
MyGenClass<String, String> obj = new MyGenClass<String, String>()
```

In this case, both **T** and **V** will be replaced by **String**. This makes both versions of **set()** identical, which is, of course, an error.

The second and more fundamental problem is that the type erasure of **set()** reduces both versions to the following:

```
void set(Object o) { // ...}
```

Thus, the overloading of **set()** as attempted in **MyGenClass** is inherently ambiguous.

Ambiguity errors can be tricky to fix. For example, if you know that **V** will always be some type of **Number**, you might try to fix **MyGenClass** by rewriting its declaration as shown here:

```
class MyGenClass<T, V extends Number> { // almost OK!}
```

This change causes **MyGenClass** to compile, and you can even instantiate objects like the one shown here:

```
MyGenClass<String, Number> x = new MyGenClass<String, Number>();
```

This works because Java can accurately determine which method to call. However, ambiguity returns when you try this line:

```
MyGenClass<Number, Number> x = new MyGenClass<Number, Number>();
```

In this case, since both **T** and **V** are **Number**, which version of **set()** is to be called? The call to **set()** is now ambiguous.

Frankly, in the preceding example, it would be much better to use two separate method names, rather than trying to overload **set()**. Often, the solution to ambiguity involves the restructuring of the code, because ambiguity frequently means that you have a conceptual error in your design.

Some Generic Restrictions

There are a few restrictions that you need to keep in mind when using generics. They involve creating objects of a type parameter, static members, exceptions, and arrays. Each is examined here.

Type Parameters Can't Be Instantiated

It is not possible to create an instance of a type parameter. For example, consider this class:

```
// Can't create an instance of T.
class Gen<T> {
    T ob;

    Gen() {
        ob = new T(); // Illegal!!!
    }
}
```

Here, it is illegal to attempt to create an instance of **T**. The reason should be easy to understand: the compiler does not know what type of object to create. **T** is simply a placeholder.

Restrictions on Static Members

No **static** member can use a type parameter declared by the enclosing class. For example, both of the **static** members of this class are illegal:

```
class Wrong<T> {
    // Wrong, no static variables of type T.
    static T ob;

    // Wrong, no static method can use T.
    static T getob() {
        return ob;
    }
}
```

Although you can't declare **static** members that use a type parameter declared by the enclosing class, you can declare **static** generic methods, which define their own type parameters, as was done earlier in this chapter.

Generic Array Restrictions

There are two important generics restrictions that apply to arrays. First, you cannot instantiate an array whose element type is a type parameter. Second, you cannot create an array of type-specific generic references. The following short program shows both situations:

```
// Generics and arrays.
class Gen<T extends Number> {
    T ob;
```

```

T vals[]; // OK

Gen(T o, T[] nums) {
    ob = o;

    // This statement is illegal.
    // vals = new T[10]; // can't create an array of T

    // But, this statement is OK.
    vals = nums; // OK to assign reference to existent array
}
}

class GenArrays {
    public static void main(String args[]) {
        Integer n[] = { 1, 2, 3, 4, 5 };

        Gen<Integer> iOb = new Gen<Integer>(50, n);

        // Can't create an array of type-specific generic references.
        // Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!

        // This is OK.
        Gen<?> gens[] = new Gen<?>[10]; // OK
    }
}

```

As the program shows, it's valid to declare a reference to an array of type **T**, as this line does:

```
T vals[]; // OK
```

But, you cannot instantiate an array of **T**, as this commented-out line attempts:

```
// vals = new T[10]; // can't create an array of T
```

The reason you can't create an array of **T** is that there is no way for the compiler to know what type of array to actually create.

However, you can pass a reference to a type-compatible array to **Gen()** when an object is created and assign that reference to **vals**, as the program does in this line:

```
vals = nums; // OK to assign reference to existent array
```

This works because the array passed to **Gen** has a known type, which will be the same type as **T** at the time of object creation.

Inside **main()**, notice that you can't declare an array of references to a specific generic type. That is, this line

```
// Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!
```

won't compile.

You *can* create an array of references to a generic type if you use a wildcard, however, as shown here:

```
Gen<?> gens[] = new Gen<?>[10]; // OK
```

This approach is better than using an array of raw types, because at least some type checking will still be enforced.

Generic Exception Restriction

A generic class cannot extend **Throwable**. This means that you cannot create generic exception classes.

This page has been intentionally left blank

CHAPTER

15

Lambda Expressions

During Java's ongoing development and evolution, many features have been added since its original 1.0 release. However, two stand out because they have profoundly reshaped the language, fundamentally changing the way that code is written. The first was the addition of generics, added by JDK 5. (See Chapter 14.) The second is the *lambda expression*, which is the subject of this chapter.

Added by JDK 8, lambda expressions (and their related features) significantly enhance Java because of two primary reasons. First, they add new syntax elements that increase the expressive power of the language. In the process, they streamline the way that certain common constructs are implemented. Second, the addition of lambda expressions resulted in new capabilities being incorporated into the API library. Among these new capabilities are the ability to more easily take advantage of the parallel processing capabilities of multi-core environments, especially as it relates to the handling of for-each style operations, and the new stream API, which supports pipeline operations on data. The addition of lambda expressions also provided the catalyst for other new Java features, including the default method (described in Chapter 9), which lets you define default behavior for an interface method, and the method reference (described here), which lets you refer to a method without executing it.

Beyond the benefits that lambda expressions bring to the language, there is another reason why they constitute an important addition to Java. Over the past few years, lambda expressions have become a major focus of computer language design. For example, they have been added to languages such as C# and C++. Their inclusion in JDK 8 helps Java remain the vibrant, innovative language that programmers have come to expect.

In the final analysis, in much the same way that generics reshaped Java several years ago, lambda expressions are reshaping Java today. Simply put, lambda expressions will impact virtually all Java programmers. They truly are that important.

Introducing Lambda Expressions

Key to understanding Java's implementation of lambda expressions are two constructs. The first is the lambda expression, itself. The second is the functional interface. Let's begin with a simple definition of each.

A *lambda expression* is, essentially, an anonymous (that is, unnamed) method. However, this method is not executed on its own. Instead, it is used to implement a method defined by a functional interface. Thus, a lambda expression results in a form of anonymous class. Lambda expressions are also commonly referred to as *closures*.

A *functional interface* is an interface that contains one and only one abstract method. Normally, this method specifies the intended purpose of the interface. Thus, a functional interface typically represents a single action. For example, the standard interface **Runnable** is a functional interface because it defines only one method: **run()**. Therefore, **run()** defines the action of **Runnable**. Furthermore, a functional interface defines the *target type* of a lambda expression. Here is a key point: a lambda expression can be used only in a context in which its target type is specified. One other thing: a functional interface is sometimes referred to as a *SAM type*, where SAM stands for Single Abstract Method.

NOTE A functional interface may specify any public method defined by **Object**, such as **equals()**, without affecting its "functional interface" status. The public **Object** methods are considered implicit members of a functional interface because they are automatically implemented by an instance of a functional interface.

Let's now look more closely at both lambda expressions and functional interfaces.

Lambda Expression Fundamentals

The lambda expression introduces a new syntax element and operator into the Java language. The new operator, sometimes referred to as the *lambda operator* or the *arrow operator*, is **->**. It divides a lambda expression into two parts. The left side specifies any parameters required by the lambda expression. (If no parameters are needed, an empty parameter list is used.) On the right side is the *lambda body*, which specifies the actions of the lambda expression. The **->** can be verbalized as "becomes" or "goes to."

Java defines two types of lambda bodies. One consists of a single expression, and the other type consists of a block of code. We will begin with lambdas that define a single expression. Lambdas with block bodies are discussed later in this chapter.

At this point, it will be helpful to look a few examples of lambda expressions before continuing. Let's begin with what is probably the simplest type of lambda expression you can write. It evaluates to a constant value and is shown here:

```
() -> 123.45
```

This lambda expression takes no parameters, thus the parameter list is empty. It returns the constant value 123.45. Therefore, it is similar to the following method:

```
double myMeth() { return 123.45; }
```

Of course, the method defined by a lambda expression does not have a name.

A slightly more interesting lambda expression is shown here:

```
() -> Math.random() * 100
```

This lambda expression obtains a pseudo-random value from **Math.random()**, multiplies it by 100, and returns the result. It, too, does not require a parameter.

When a lambda expression requires a parameter, it is specified in the parameter list on the left side of the lambda operator. Here is a simple example:

```
(n) -> (n % 2) == 0
```

This lambda expression returns **true** if the value of parameter **n** is even. Although it is possible to explicitly specify the type of a parameter, such as **n** in this case, often you won't need to do so because in many cases its type can be inferred. Like a named method, a lambda expression can specify as many parameters as needed.

Functional Interfaces

As stated, a functional interface is an interface that specifies only one abstract method. If you have been programming in Java for some time, you might at first think that all interface methods are implicitly abstract. Although this was true prior to JDK 8, the situation has changed. As explained in Chapter 9, beginning with JDK 8, it is possible to specify default behavior for a method declared in an interface. This is called a *default method*. Today, an interface method is abstract only if it does not specify a default implementation. Because nondefault interface methods are implicitly abstract, there is no need to use the **abstract** modifier (although you can specify it, if you like).

Here is an example of a functional interface:

```
interface MyNumber {  
    double getValue();  
}
```

In this case, the method **getValue()** is implicitly abstract, and it is the only method defined by **MyNumber**. Thus, **MyNumber** is a functional interface, and its function is defined by **getValue()**.

As mentioned earlier, a lambda expression is not executed on its own. Rather, it forms the implementation of the abstract method defined by the functional interface that specifies its target type. As a result, a lambda expression can be specified only in a context in which a target type is defined. One of these contexts is created when a lambda expression is assigned to a functional interface reference. Other target type contexts include variable initialization, **return** statements, and method arguments, to name a few.

Let's work through an example that shows how a lambda expression can be used in an assignment context. First, a reference to the functional interface **MyNumber** is declared:

```
// Create a reference to a MyNumber instance.  
MyNumber myNum;
```

Next, a lambda expression is assigned to that interface reference:

```
// Use a lambda in an assignment context.  
myNum = () -> 123.45;
```

When a lambda expression occurs in a target type context, an instance of a class is automatically created that implements the functional interface, with the lambda expression defining the behavior of the abstract method declared by the functional interface. When that method is called through the target, the lambda expression is executed. Thus, a lambda expression gives us a way to transform a code segment into an object.

In the preceding example, the lambda expression becomes the implementation for the `getValue()` method. As a result, the following displays the value 123.45:

```
// Call getValue(), which is implemented by the previously assigned  
// lambda expression.  
System.out.println("myNum.getValue());
```

Because the lambda expression assigned to `myNum` returns the value 123.45, that is the value obtained when `getValue()` is called.

In order for a lambda expression to be used in a target type context, the type of the abstract method and the type of the lambda expression must be compatible. For example, if the abstract method specifies two `int` parameters, then the lambda must specify two parameters whose type either is explicitly `int` or can be implicitly inferred as `int` by the context. In general, the type and number of the lambda expression's parameters must be compatible with the method's parameters; the return types must be compatible; and any exceptions thrown by the lambda expression must be acceptable to the method.

Some Lambda Expression Examples

With the preceding discussion in mind, let's look at some simple examples that illustrate the basic lambda expression concepts. The first example puts together the pieces shown in the foregoing section.

```
// Demonstrate a simple lambda expression.  
  
// A functional interface.  
interface MyNumber {  
    double getValue();  
}  
  
class LambdaDemo {  
    public static void main(String args[])  
    {  
        MyNumber myNum; // declare an interface reference  
  
        // Here, the lambda expression is simply a constant expression.  
        // When it is assigned to myNum, a class instance is  
        // constructed in which the lambda expression implements  
        // the getValue() method in MyNumber.  
        myNum = () -> 123.45;
```

```
// Call getValue(), which is provided by the previously assigned
// lambda expression.
System.out.println("A fixed value: " + myNum.getValue());

// Here, a more complex expression is used.
myNum = () -> Math.random() * 100;

// These call the lambda expression in the previous line.
System.out.println("A random value: " + myNum.getValue());
System.out.println("Another random value: " + myNum.getValue());

// A lambda expression must be compatible with the method
// defined by the functional interface. Therefore, this won't work:
// myNum = () -> "123.03"; // Error!
}
```

Sample output from the program is shown here:

```
A fixed value: 123.45
A random value: 88.90663650412304
Another random value: 53.00582701784129
```

As mentioned, the lambda expression must be compatible with the abstract method that it is intended to implement. For this reason, the commented-out line at the end of the preceding program is illegal because a value of type **String** is not compatible with **double**, which is the return type required by **getValue()**.

The next example shows the use of a parameter with a lambda expression:

```
// Demonstrate a lambda expression that takes a parameter.

// Another functional interface.
interface NumericTest {
    boolean test(int n);
}

class LambdaDemo2 {
    public static void main(String args[])
    {
        // A lambda expression that tests if a number is even.
        NumericTest isEven = (n) -> (n % 2)==0;

        if(isEven.test(10)) System.out.println("10 is even");
        if(!isEven.test(9)) System.out.println("9 is not even");

        // Now, use a lambda expression that tests if a number
        // is non-negative.
        NumericTest isNonNeg = (n) -> n >= 0;

        if(isNonNeg.test(1)) System.out.println("1 is non-negative");
        if(!isNonNeg.test(-1)) System.out.println("-1 is negative");
    }
}
```

The output from this program is shown here:

```
10 is even
9 is not even
1 is non-negative
-1 is negative
```

This program demonstrates a key fact about lambda expressions that warrants close examination. Pay special attention to the lambda expression that performs the test for evenness. It is shown again here:

```
(n) -> (n % 2) == 0
```

Notice that the type of **n** is not specified. Rather, its type is inferred from the context. In this case, its type is inferred from the parameter type of **test()** as defined by the **NumericTest** interface, which is **int**. It is also possible to explicitly specify the type of a parameter in a lambda expression. For example, this is also a valid way to write the preceding:

```
(int n) -> (n % 2) == 0
```

Here, **n** is explicitly specified as **int**. Usually it is not necessary to explicitly specify the type, but you can in those situations that require it.

This program demonstrates another important point about lambda expressions: A functional interface reference can be used to execute any lambda expression that is compatible with it. Notice that the program defines two different lambda expressions that are compatible with the **test()** method of the functional interface **NumericTest**. The first, called **isEven**, determines if a value is even. The second, called **isNonNeg**, checks if a value is non-negative. In each case, the value of the parameter **n** is tested. Because each lambda expression is compatible with **test()**, each can be executed through a **NumericTest** reference.

One other point before moving on. When a lambda expression has only one parameter, it is not necessary to surround the parameter name with parentheses when it is specified on the left side of the lambda operator. For example, this is also a valid way to write the lambda expression used in the program:

```
n -> (n % 2) == 0
```

For consistency, this book will surround all lambda expression parameter lists with parentheses, even those containing only one parameter. Of course, you are free to adopt a different style.

The next program demonstrates a lambda expression that takes two parameters. In this case, the lambda expression tests if one number is a factor of another.

```
// Demonstrate a lambda expression that takes two parameters.

interface NumericTest2 {
    boolean test(int n, int d);
}

class LambdaDemo3 {
```

```

public static void main(String args[])
{
    // This lambda expression determines if one number is
    // a factor of another.
    NumericTest2 isFactor = (n, d) -> (n % d) == 0;

    if(isFactor.test(10, 2))
        System.out.println("2 is a factor of 10");

    if(!isFactor.test(10, 3))
        System.out.println("3 is not a factor of 10");
}

```

The output is shown here:

```

2 is a factor of 10
3 is not a factor of 10

```

In this program, the functional interface **NumericTest2** defines the **test()** method:

```
boolean test(int n, int d);
```

In this version, **test()** specifies two parameters. Thus, for a lambda expression to be compatible with **test()**, the lambda expression must also specify two parameters. Notice how they are specified:

```
(n, d) -> (n % d) == 0
```

The two parameters, **n** and **d**, are specified in the parameter list, separated by commas. This example can be generalized. Whenever more than one parameter is required, the parameters are specified, separated by commas, in a parenthesized list on the left side of the lambda operator.

Here is an important point about multiple parameters in a lambda expression: If you need to explicitly declare the type of a parameter, then all of the parameters must have declared types. For example, this is legal:

```
(int n, int d) -> (n % d) == 0
```

But this is not:

```
(int n, d) -> (n % d) == 0
```

Block Lambda Expressions

The body of the lambdas shown in the preceding examples consist of a single expression. These types of lambda bodies are referred to as *expression bodies*, and lambdas that have expression bodies are sometimes called *expression lambdas*. In an expression body, the code on the right side of the lambda operator must consist of a single expression. While

expression lambdas are quite useful, sometimes the situation will require more than a single expression. To handle such cases, Java supports a second type of lambda expression in which the code on the right side of the lambda operator consists of a block of code that can contain more than one statement. This type of lambda body is called a *block body*. Lambdas that have block bodies are sometimes referred to as *block lambdas*.

A block lambda expands the types of operations that can be handled within a lambda expression because it allows the body of the lambda to contain multiple statements. For example, in a block lambda you can declare variables, use loops, specify **if** and **switch** statements, create nested blocks, and so on. A block lambda is easy to create. Simply enclose the body within braces as you would any other block of statements.

Aside from allowing multiple statements, block lambdas are used much like the expression lambdas just discussed. One key difference, however, is that you must explicitly use a **return** statement to return a value. This is necessary because a block lambda body does not represent a single expression.

Here is an example that uses a block lambda to compute and return the factorial of an **int** value:

```
// A block lambda that computes the factorial of an int value.

interface NumericFunc {
    int func(int n);
}

class BlockLambdaDemo {
    public static void main(String args[])
    {

        // This block lambda computes the factorial of an int value.
        NumericFunc factorial = (n) -> {
            int result = 1;

            for(int i=1; i <= n; i++)
                result = i * result;

            return result;
        };

        System.out.println("The factorial of 3 is " + factorial.func(3));
        System.out.println("The factorial of 5 is " + factorial.func(5));
    }
}
```

The output is shown here:

```
The factorial of 3 is 6
The factorial of 5 is 120
```

In the program, notice that the block lambda declares a variable called **result**, uses a **for** loop, and has a **return** statement. These are legal inside a block lambda body. In essence, the block body of a lambda is similar to a method body. One other point. When a **return**

statement occurs within a lambda expression, it simply causes a return from the lambda. It does not cause an enclosing method to return.

Another example of a block lambda is shown in the following program. It reverses the characters in a string.

```
// A block lambda that reverses the characters in a string.

interface StringFunc {
    String func(String n);
}

class BlockLambdaDemo2 {
    public static void main(String args[])
    {

        // This block lambda reverses the characters in a string.
        StringFunc reverse = (str) -> {
            String result = "";
            int i;

            for(i = str.length()-1; i >= 0; i--)
                result += str.charAt(i);

            return result;
        };

        System.out.println("Lambda reversed is " +
                           reverse.func("Lambda"));
        System.out.println("Expression reversed is " +
                           reverse.func("Expression"));
    }
}
```

The output is shown here:

```
Lambda reversed is adbmaL
Expression reversed is noisserpxE
```

In this example, the functional interface **StringFunc** declares the **func()** method. This method takes a parameter of type **String** and has a return type of **String**. Thus, in the **reverse** lambda expression, the type of **str** is inferred to be **String**. Notice that the **charAt()** method is called on **str**. This is legal because of the inference that **str** is of type **String**.

Generic Functional Interfaces

A lambda expression, itself, cannot specify type parameters. Thus, a lambda expression cannot be generic. (Of course, because of type inference, all lambda expressions exhibit some “generic-like” qualities.) However, the functional interface associated with a lambda expression can be generic. In this case, the target type of the lambda expression is

determined, in part, by the type argument or arguments specified when a functional interface reference is declared.

To understand the value of generic functional interfaces, consider this. The two examples in the previous section used two different functional interfaces, one called **NumericFunc** and the other called **StringFunc**. However, both defined a method called **func()** that took one parameter and returned a result. In the first case, the type of the parameter and return type was **int**. In the second case, the parameter and return type was **String**. Thus, the only difference between the two methods was the type of data they required. Instead of having two functional interfaces whose methods differ only in their data types, it is possible to declare one generic interface that can be used to handle both circumstances. The following program shows this approach:

```
// Use a generic functional interface with lambda expressions.

// A generic functional interface.
interface SomeFunc<T> {
    T func(T t);
}

class GenericFunctionalInterfaceDemo {
    public static void main(String args[])
    {

        // Use a String-based version of SomeFunc.
        SomeFunc<String> reverse = (str) -> {
            String result = "";
            int i;

            for(i = str.length()-1; i >= 0; i--)
                result += str.charAt(i);

            return result;
        };

        System.out.println("Lambda reversed is " +
                           reverse.func("Lambda"));
        System.out.println("Expression reversed is " +
                           reverse.func("Expression"));

        // Now, use an Integer-based version of SomeFunc.
        SomeFunc<Integer> factorial = (n) -> {
            int result = 1;

            for(int i=1; i <= n; i++)
                result = i * result;

            return result;
        };

        System.out.println("The factorial of 3 is " + factorial.func(3));
        System.out.println("The factorial of 5 is " + factorial.func(5));
    }
}
```

The output is shown here:

```
Lambda reversed is adbmaL
Expression reversed is noisserpxE
The factorial of 3 is 6
The factorial of 5 is 120
```

In the program, the generic functional interface **SomeFunc** is declared as shown here:

```
interface SomeFunc<T> {
    T func(T t);
}
```

Here, **T** specifies both the return type and the parameter type of **func()**. This means that it is compatible with any lambda expression that takes one parameter and returns a value of the same type.

The **SomeFunc** interface is used to provide a reference to two different types of lambdas. The first uses type **String**. The second uses type **Integer**. Thus, the same functional interface can be used to refer to the **reverse** lambda and the **factorial** lambda. Only the type argument passed to **SomeFunc** differs.

Passing Lambda Expressions as Arguments

As explained earlier, a lambda expression can be used in any context that provides a target type. One of these is when a lambda expression is passed as an argument. In fact, passing a lambda expression as an argument is a common use of lambdas. Moreover, it is a very powerful use because it gives you a way to pass executable code as an argument to a method. This greatly enhances the expressive power of Java.

To pass a lambda expression as an argument, the type of the parameter receiving the lambda expression argument must be of a functional interface type compatible with the lambda. Although using a lambda expression as an argument is straightforward, it is still helpful to see it in action. The following program demonstrates the process:

```
// Use lambda expressions as an argument to a method.

interface StringFunc {
    String func(String n);
}

class LambdasAsArgumentsDemo {

    // This method has a functional interface as the type of
    // its first parameter. Thus, it can be passed a reference to
    // any instance of that interface, including the instance created
    // by a lambda expression.
    // The second parameter specifies the string to operate on.
    static String stringOp(StringFunc sf, String s) {
        return sf.func(s);
    }

    public static void main(String args[])
}
```

```

{
    String inStr = "Lambdas add power to Java";
    String outStr;

    System.out.println("Here is input string: " + inStr);

    // Here, a simple expression lambda that uppercases a string
    // is passed to stringOp( ).
    outStr = stringOp((str) -> str.toUpperCase(), inStr);
    System.out.println("The string in uppercase: " + outStr);

    // This passes a block lambda that removes spaces.
    outStr = stringOp((str) -> {
        String result = "";
        int i;

        for(i = 0; i < str.length(); i++)
            if(str.charAt(i) != ' ')
                result += str.charAt(i);

        return result;
    }, inStr);

    System.out.println("The string with spaces removed: " + outStr);

    // Of course, it is also possible to pass a StringFunc instance
    // created by an earlier lambda expression. For example,
    // after this declaration executes, reverse refers to an
    // instance of StringFunc.
    StringFunc reverse = (str) -> {
        String result = "";
        int i;

        for(i = str.length()-1; i >= 0; i--)
            result += str.charAt(i);

        return result;
    };

    // Now, reverse can be passed as the first parameter to stringOp()
    // since it refers to a StringFunc object.
    System.out.println("The string reversed: " +
                       stringOp(reverse, inStr));
}
}

```

The output is shown here:

```

Here is input string: Lambdas add power to Java
The string in uppercase: LAMBDAS ADD POWER TO JAVA
The string with spaces removed: LambdasaddpowertoJava
The string reversed: avaJ ot rewop dda sadbmaL

```

In the program, first notice the **stringOp()** method. It has two parameters. The first is of type **StringFunc**, which is a functional interface. Thus, this parameter can receive a reference to any instance of **StringFunc**, including one created by a lambda expression. The second argument of **stringOp()** is of type **String**, and this is the string operated on.

Next, notice the first call to **stringOp()**, shown again here:

```
outStr = stringOp((str) -> str.toUpperCase(), inStr);
```

Here, a simple expression lambda is passed as an argument. When this occurs, an instance of the functional interface **StringFunc** is created and a reference to that object is passed to the first parameter of **stringOp()**. Thus, the lambda code, embedded in a class instance, is passed to the method. The target type context is determined by the type of parameter. Because the lambda expression is compatible with that type, the call is valid. Embedding simple lambdas, such as the one just shown, inside a method call is often a convenient technique—especially when the lambda expression is intended for a single use.

Next, the program passes a block lambda to **stringOp()**. This lambda removes spaces from a string. It is shown again here:

```
outStr = stringOp((str) -> {
    String result = "";
    int i;

    for(i = 0; i < str.length(); i++)
        if(str.charAt(i) != ' ')
            result += str.charAt(i);

    return result;
}, inStr);
```

Although this uses a block lambda, the process of passing the lambda expression is the same as just described for the simple expression lambda. In this case, however, some programmers will find the syntax a bit awkward.

When a block lambda seems overly long to embed in a method call, it is an easy matter to assign that lambda to a functional interface variable, as the previous examples have done. Then, you can simply pass that reference to the method. This technique is shown at the end of the program. There, a block lambda is defined that reverses a string. This lambda is assigned to **reverse**, which is a reference to a **StringFunc** instance. Thus, **reverse** can be used as an argument to the first parameter of **stringOp()**. The program then calls **stringOp()**, passing in **reverse** and the string on which to operate. Because the instance obtained by the evaluation of each lambda expression is an implementation of **StringFunc**, each can be used as the first parameter to **stringOp()**.

One last point: In addition to variable initialization, assignment, and argument passing, the following also constitute target type contexts: casts, the **? operator**, array initializers, **return** statements, and lambda expressions, themselves.

Lambda Expressions and Exceptions

A lambda expression can throw an exception. However, if it throws a checked exception, then that exception must be compatible with the exception(s) listed in the **throws** clause of the abstract method in the functional interface. Here is an example that illustrates this fact. It computes the average of an array of **double** values. If a zero-length array is passed, however, it throws the custom exception **EmptyArrayException**. As the example shows, this exception is listed in the **throws** clause of **func()** declared inside the **DoubleNumericArrayFunc** functional interface.

```
// Throw an exception from a lambda expression.

interface DoubleNumericArrayFunc {
    double func(double[] n) throws EmptyArrayException;
}

class EmptyArrayException extends Exception {
    EmptyArrayException() {
        super("Array Empty");
    }
}

class LambdaExceptionDemo {

    public static void main(String args[]) throws EmptyArrayException
    {
        double[] values = { 1.0, 2.0, 3.0, 4.0 };

        // This block lambda computes the average of an array of doubles.
        DoubleNumericArrayFunc average = (n) -> {
            double sum = 0;

            if(n.length == 0)
                throw new EmptyArrayException();

            for(int i=0; i < n.length; i++)
                sum += n[i];

            return sum / n.length;
        };

        System.out.println("The average is " + average.func(values));

        // This causes an exception to be thrown.
        System.out.println("The average is " + average.func(new double[0]));
    }
}
```

The first call to **average.func()** returns the value 2.5. The second call, which passes a zero-length array, causes an **EmptyArrayException** to be thrown. Remember, the inclusion of the **throws** clause in **func()** is necessary. Without it, the program will not compile because the lambda expression will no longer be compatible with **func()**.

This example demonstrates another important point about lambda expressions. Notice that the parameter specified by `func()` in the functional interface `DoubleNumericArrayFunc` is an array. However, the parameter to the lambda expression is simply `n`, rather than `n[]`. Remember, the type of a lambda expression parameter will be inferred from the target context. In this case, the target context is `double[]`, thus the type of `n` will be `double[]`. It is not necessary (or legal) to specify it as `n[]`. It would be legal to explicitly declare it as `double[] n`, but doing so gains nothing in this case.

Lambda Expressions and Variable Capture

Variables defined by the enclosing scope of a lambda expression are accessible within the lambda expression. For example, a lambda expression can use an instance or `static` variable defined by its enclosing class. A lambda expression also has access to `this` (both explicitly and implicitly), which refers to the invoking instance of the lambda expression's enclosing class. Thus, a lambda expression can obtain or set the value of an instance or `static` variable and call a method defined by its enclosing class.

However, when a lambda expression uses a local variable from its enclosing scope, a special situation is created that is referred to as a *variable capture*. In this case, a lambda expression may only use local variables that are *effectively final*. An effectively final variable is one whose value does not change after it is first assigned. There is no need to explicitly declare such a variable as `final`, although doing so would not be an error. (The `this` parameter of an enclosing scope is automatically effectively final, and lambda expressions do not have a `this` of their own.)

It is important to understand that a local variable of the enclosing scope cannot be modified by the lambda expression. Doing so would remove its effectively final status, thus rendering it illegal for capture.

The following program illustrates the difference between effectively final and mutable local variables:

```
// An example of capturing a local variable from the enclosing scope.

interface MyFunc {
    int func(int n);
}

class VarCapture {
    public static void main(String args[])
    {
        // A local variable that can be captured.
        int num = 10;

        MyFunc myLambda = (n) -> {
            // This use of num is OK. It does not modify num.
            int v = num + n;

            // However, the following is illegal because it attempts
            // to modify the value of num.
            // num++;
        };
    }
}
```

```

        return v;
    };

    // The following line would also cause an error, because
    // it would remove the effectively final status from num.
//    num = 9;
}
}

```

As the comments indicate, `num` is effectively final and can, therefore, be used inside `myLambda`. However, if `num` were to be modified, either inside the lambda or outside of it, `num` would lose its effectively final status. This would cause an error, and the program would not compile.

It is important to emphasize that a lambda expression can use and modify an instance variable from its invoking class. It just can't use a local variable of its enclosing scope unless that variable is effectively final.

Method References

There is an important feature related to lambda expressions called the *method reference*. A method reference provides a way to refer to a method without executing it. It relates to lambda expressions because it, too, requires a target type context that consists of a compatible functional interface. When evaluated, a method reference also creates an instance of the functional interface.

There are different types of method references. We will begin with method references to **static** methods.

Method References to static Methods

To create a **static** method reference, use this general syntax:

ClassName::methodName

Notice that the class name is separated from the method name by a double colon. The `::` is a new separator that has been added to Java by JDK 8 expressly for this purpose. This method reference can be used anywhere in which it is compatible with its target type.

The following program demonstrates a **static** method reference:

```

// Demonstrate a method reference for a static method.

// A functional interface for string operations.
interface StringFunc {
    String func(String n);
}

// This class defines a static method called strReverse().
class MyStringOps {
    // A static method that reverses a string.
    static String strReverse(String str) {
        String result = "";

```

```

        int i;

        for(i = str.length()-1; i >= 0; i--)
            result += str.charAt(i);

        return result;
    }

}

class MethodRefDemo {

    // This method has a functional interface as the type of
    // its first parameter. Thus, it can be passed any instance
    // of that interface, including a method reference.
    static String stringOp(StringFunc sf, String s) {
        return sf.func(s);
    }

    public static void main(String args[])
    {
        String inStr = "Lambdas add power to Java";
        String outStr;

        // Here, a method reference to strReverse is passed to stringOp().
        outStr = stringOp(MyStringOps::strReverse, inStr);

        System.out.println("Original string: " + inStr);
        System.out.println("String reversed: " + outStr);
    }
}

```

The output is shown here:

```

Original string: Lambdas add power to Java
String reversed: avaJ ot rewop dda sadbmaL

```

In the program, pay special attention to this line:

```
outStr = stringOp(MyStringOps::strReverse, inStr);
```

Here, a reference to the **static** method **strReverse()**, declared inside **MyStringOps**, is passed as the first argument to **stringOp()**. This works because **strReverse** is compatible with the **StringFunc** functional interface. Thus, the expression **MyStringOps::strReverse** evaluates to a reference to an object in which **strReverse** provides the implementation of **func()** in **StringFunc**.

Method References to Instance Methods

To pass a reference to an instance method on a specific object, use this basic syntax:

objRef:methodName

As you can see, the syntax is similar to that used for a **static** method, except that an object reference is used instead of a class name. Here is the previous program rewritten to use an instance method reference:

```
// Demonstrate a method reference to an instance method

// A functional interface for string operations.
interface StringFunc {
    String func(String n);
}

// Now, this class defines an instance method called strReverse().
class MyStringOps {
    String strReverse(String str) {
        String result = "";
        int i;

        for(i = str.length()-1; i >= 0; i--)
            result += str.charAt(i);

        return result;
    }
}

class MethodRefDemo2 {

    // This method has a functional interface as the type of
    // its first parameter. Thus, it can be passed any instance
    // of that interface, including method references.
    static String stringOp(StringFunc sf, String s) {
        return sf.func(s);
    }

    public static void main(String args[])
    {
        String inStr = "Lambdas add power to Java";
        String outStr;

        // Create a MyStringOps object.
        MyStringOps strOps = new MyStringOps( );

        // Now, a method reference to the instance method strReverse
        // is passed to stringOp().
        outStr = stringOp(strOps::strReverse, inStr);

        System.out.println("Original string: " + inStr);
        System.out.println("String reversed: " + outStr);
    }
}
```

This program produces the same output as the previous version.

In the program, notice that **strReverse()** is now an instance method of **MyStringOps**. Inside **main()**, an instance of **MyStringOps** called **strOps** is created. This instance is used to create the method reference to **strReverse** in the call to **stringOp**, as shown again, here:

```
outStr = stringOp(strOps::strReverse, inStr);
```

In this example, **strReverse()** is called on the **strOps** object.

It is also possible to handle a situation in which you want to specify an instance method that can be used with any object of a given class—not just a specified object. In this case, you will create a method reference as shown here:

ClassName::instanceMethodName

Here, the name of the class is used instead of a specific object, even though an instance method is specified. With this form, the first parameter of the functional interface matches the invoking object and the second parameter matches the parameter specified by the method. Here is an example. It defines a method called **counter()** that counts the number of objects in an array that satisfy the condition defined by the **func()** method of the **MyFunc** functional interface. In this case, it counts instances of the **HighTemp** class.

```
// Use an instance method reference with different objects.

// A functional interface that takes two reference arguments
// and returns a boolean result.
interface MyFunc<T> {
    boolean func(T v1, T v2);
}

// A class that stores the temperature high for a day.
class HighTemp {
    private int hTemp;

    HighTemp(int ht) { hTemp = ht; }

    // Return true if the invoking HighTemp object has the same
    // temperature as ht2.
    boolean sameTemp(HighTemp ht2) {
        return hTemp == ht2.hTemp;
    }

    // Return true if the invoking HighTemp object has a temperature
    // that is less than ht2.
    boolean lessThanTemp(HighTemp ht2) {
        return hTemp < ht2.hTemp;
    }
}

class InstanceMethWithObjectRefDemo {

    // A method that returns the number of occurrences
    // of an object for which some criteria, as specified by
    // the MyFunc parameter, is true.
    static <T> int counter(T[] vals, MyFunc<T> f, T v) {
```

```

int count = 0;

for(int i=0; i < vals.length; i++)
    if(f.func(vals[i], v)) count++;

return count;
}

public static void main(String args[])
{
    int count;

    // Create an array of HighTemp objects.
    HighTemp[] weekDayHighs = { new HighTemp(89), new HighTemp(82),
                               new HighTemp(90), new HighTemp(89),
                               new HighTemp(89), new HighTemp(91),
                               new HighTemp(84), new HighTemp(83) };

    // Use counter() with arrays of the class HighTemp.
    // Notice that a reference to the instance method
    // sameTemp() is passed as the second argument.
    count = counter(weekDayHighs, HighTemp::sameTemp,
                    new HighTemp(89));
    System.out.println(count + " days had a high of 89");

    // Now, create and use another array of HighTemp objects.
    HighTemp[] weekDayHighs2 = { new HighTemp(32), new HighTemp(12),
                                new HighTemp(24), new HighTemp(19),
                                new HighTemp(18), new HighTemp(12),
                                new HighTemp(-1), new HighTemp(13) };

    count = counter(weekDayHighs2, HighTemp::sameTemp,
                    new HighTemp(12));
    System.out.println(count + " days had a high of 12");

    // Now, use lessThanTemp() to find days when temperature was less
    // than a specified value.
    count = counter(weekDayHighs, HighTemp::lessThanTemp,
                    new HighTemp(89));
    System.out.println(count + " days had a high less than 89");

    count = counter(weekDayHighs2, HighTemp::lessThanTemp,
                    new HighTemp(19));
    System.out.println(count + " days had a high of less than 19");
}
}

```

The output is shown here:

```

3 days had a high of 89
2 days had a high of 12
3 days had a high less than 89
5 days had a high of less than 19

```

In the program, notice that **HighTemp** has two instance methods: **sameTemp()** and **lessThanTemp()**. The first returns **true** if two **HighTemp** objects contain the same temperature. The second returns **true** if the temperature of the invoking object is less than that of the passed object. Each method has a parameter of type **HighTemp** and each method returns a **boolean** result. Thus, each is compatible with the **MyFunc** functional interface because the invoking object type can be mapped to the first parameter of **func()** and the argument mapped to **func()**'s second parameter. Thus, when the expression

```
HighTemp::sameTemp
```

is passed to the **counter()** method, an instance of the functional interface **MyFunc** is created in which the parameter type of the first parameter is that of the invoking object of the instance method, which is **HighTemp**. The type of the second parameter is also **HighTemp** because that is the type of the parameter to **sameTemp()**. The same is true for the **lessThanTemp()** method.

One other point: you can refer to the superclass version of a method by use of **super**, as shown here:

```
super::name
```

The name of the method is specified by *name*.

Method References with Generics

You can use method references with generic classes and/or generic methods. For example, consider the following program:

```
// Demonstrate a method reference to a generic method
// declared inside a non-generic class.

// A functional interface that operates on an array
// and a value, and returns an int result.
interface MyFunc<T> {
    int func(T[] vals, T v);
}

// This class defines a method called countMatching() that
// returns the number of items in an array that are equal
// to a specified value. Notice that countMatching()
// is generic, but MyArrayOps is not.
class MyArrayOps {
    static <T> int countMatching(T[] vals, T v) {
        int count = 0;

        for(int i=0; i < vals.length; i++)
            if(vals[i] == v) count++;

        return count;
    }
}
```

```

class GenericMethodRefDemo {

    // This method has the MyFunc functional interface as the
    // type of its first parameter. The other two parameters
    // receive an array and a value, both of type T.
    static <T> int myOp(MyFunc<T> f, T[] vals, T v) {
        return f.func(vals, v);
    }

    public static void main(String args[])
    {
        Integer[] vals = { 1, 2, 3, 4, 2, 3, 4, 4, 5 };
        String[] strs = { "One", "Two", "Three", "Two" };
        int count;

        count = myOp(MyArrayOps::<Integer>countMatching, vals, 4);
        System.out.println("vals contains " + count + " 4s");

        count = myOp(MyArrayOps::<String>countMatching, strs, "Two");
        System.out.println("strs contains " + count + " Twos");
    }
}

```

The output is shown here:

```

vals contains 3 4s
strs contains 2 Twos

```

In the program, **MyArrayOps** is a non-generic class that contains a generic method called **countMatching()**. The method returns a count of the elements in an array that match a specified value. Notice how the generic type argument is specified. For example, its first call in **main()**, shown here:

```
count = myOp(MyArrayOps::<Integer>countMatching, vals, 4);
```

passes the type argument **Integer**. Notice that it occurs after the **::**. This syntax can be generalized: When a generic method is specified as a method reference, its type argument comes after the **::** and before the method name. It is important to point out, however, that explicitly specifying the type argument is not required in this situation (and many others) because the type argument would have been automatically inferred. In cases in which a generic class is specified, the type argument follows the class name and precedes the **::**.

Although the preceding examples show the mechanics of using method references, they don't show their real benefits. One place method references can be quite useful is in conjunction with the Collections Framework, which is described later in Chapter 18. However, for completeness, a short, but effective, example that uses a method reference to help determine the largest element in a collection is included here. (If you are unfamiliar with the Collections Framework, return to this example after you have worked through Chapter 18.)

One way to find the largest element in a collection is to use the **max()** method defined by the **Collections** class. For the version of **max()** used here, you must pass a reference to the collection and an instance of an object that implements the **Comparator<T>** interface. This interface specifies how two objects are compared. It defines only one abstract method, called **compare()**, that takes two arguments, each the type of the objects being compared. It must return greater than zero if the first argument is greater than the second, zero if the two arguments are equal, and less than zero if the first object is less than the second.

In the past, to use **max()** with user-defined objects, an instance of **Comparator<T>** had to be obtained by first explicitly implementing it by a class, and then creating an instance of that class. This instance was then passed as the comparator to **max()**. With JDK 8, it is now possible to simply pass a reference to a comparison method to **max()** because doing so automatically implements the comparator. The following simple example shows the process by creating an **ArrayList** of **MyClass** objects and then finding the one in the list that has the highest value (as defined by the comparison method).

```
// Use a method reference to help find the maximum value in a collection.
import java.util.*;

class MyClass {
    private int val;

    MyClass(int v) { val = v; }

    int getVal() { return val; }
}

class UseMethodRef {
    // A compare() method compatible with the one defined by Comparator<T>.
    static int compareMC(MyClass a, MyClass b) {
        return a.getVal() - b.getVal();
    }

    public static void main(String args[])
    {
        ArrayList<MyClass> al = new ArrayList<MyClass>();

        al.add(new MyClass(1));
        al.add(new MyClass(4));
        al.add(new MyClass(2));
        al.add(new MyClass(9));
        al.add(new MyClass(3));
        al.add(new MyClass(7));

        // Find the maximum value in al using the compareMC() method.
        MyClass maxValObj = Collections.max(al, UseMethodRef::compareMC);

        System.out.println("Maximum value is: " + maxValObj.getVal());
    }
}
```

The output is shown here:

```
Maximum value is: 9
```

In the program, notice that **MyClass** neither defines any comparison method of its own, nor does it implement **Comparator**. However, the maximum value of a list of **MyClass** items can still be obtained by calling **max()** because **UseMethodRef** defines the static method **compareMC()**, which is compatible with the **compare()** method defined by **Comparator**. Therefore, there is no need to explicitly implement and create an instance of **Comparator**.

Constructor References

Similar to the way that you can create references to methods, you can create references to constructors. Here is the general form of the syntax that you will use:

```
classname::new
```

This reference can be assigned to any functional interface reference that defines a method compatible with the constructor. Here is a simple example:

```
// Demonstrate a Constructor reference.

// MyFunc is a functional interface whose method returns
// a MyClass reference.
interface MyFunc {
    MyClass func(int n);
}

class MyClass {
    private int val;

    // This constructor takes an argument.
    MyClass(int v) { val = v; }

    // This is the default constructor.
    MyClass() { val = 0; }

    // ...

    int getVal() { return val; }
}

class ConstructorRefDemo {
    public static void main(String args[])
    {
        // Create a reference to the MyClass constructor.
        // Because func() in MyFunc takes an argument, new
        // refers to the parameterized constructor in MyClass,
        // not the default constructor.
        MyFunc myClassCons = MyClass::new;

        // Create an instance of MyClass via that constructor reference.
```

```

MyClass mc = myClassCons.func(100);

// Use the instance of MyClass just created.
System.out.println("val in mc is " + mc.getVal());
}
}

```

The output is shown here:

```
val in mc is 100
```

In the program, notice that the **func()** method of **MyFunc** returns a reference of type **MyClass** and has an **int** parameter. Next, notice that **MyClass** defines two constructors. The first specifies a parameter of type **int**. The second is the default, parameterless constructor. Now, examine the following line:

```
MyFunc myClassCons = MyClass::new;
```

Here, the expression **MyClass::new** creates a constructor reference to a **MyClass** constructor. In this case, because **MyFunc**'s **func()** method takes an **int** parameter, the constructor being referred to is **MyClass(int v)** because it is the one that matches. Also notice that the reference to this constructor is assigned to a **MyFunc** reference called **myClassCons**. After this statement executes, **myClassCons** can be used to create an instance of **MyClass**, as this line shows:

```
MyClass mc = myClassCons.func(100);
```

In essence, **myClassCons** has become another way to call **MyClass(int v)**.

Constructor references to generic classes are created in the same fashion. The only difference is that the type argument can be specified. This works the same as it does for using a generic class to create a method reference: simply specify the type argument after the class name. The following illustrates this by modifying the previous example so that **MyFunc** and **MyClass** are generic.

```

// Demonstrate a constructor reference with a generic class.

// MyFunc is now a generic functional interface.
interface MyFunc<T> {
    MyClass<T> func(T n);
}

class MyClass<T> {
    private T val;

    // A constructor that takes an argument.
    MyClass(T v) { val = v; }

    // This is the default constructor.
    MyClass() { val = null; }

    // ...
}
```

```

    T getVal() { return val; }
}

class ConstructorRefDemo2 {

    public static void main(String args[])
    {
        // Create a reference to the MyClass<T> constructor.
        MyFunc<Integer> myClassCons = MyClass<Integer>::new;

        // Create an instance of MyClass<T> via that constructor reference.
        MyClass<Integer> mc = myClassCons.func(100);

        // Use the instance of MyClass<T> just created.
        System.out.println("val in mc is " + mc.getVal());
    }
}

```

This program produces the same output as the previous version. The difference is that now both **MyFunc** and **MyClass** are generic. Thus, the sequence that creates a constructor reference can include a type argument (although one is not always needed), as shown here:

```
MyFunc<Integer> myClassCons = MyClass<Integer>::new;
```

Because the type argument **Integer** has already been specified when **myClassCons** is created, it can be used to create a **MyClass<Integer>** object, as the next line shows:

```
MyClass<Integer> mc = myClassCons.func(100);
```

Although the preceding examples demonstrate the mechanics of using a constructor reference, no one would use a constructor reference as just shown because nothing is gained. Furthermore, having what amounts to two names for the same constructor creates a confusing situation (to say the least). However, to give you the flavor of a more practical usage, the following program uses a **static** method, called **myClassFactory()**, that is a factory for objects of any type of **MyFunc** objects. It can be used to create any type of object that has a constructor compatible with its first parameter.

```

// Implement a simple class factory using a constructor reference.

interface MyFunc<R, T> {
    R func(T n);
}

// A simple generic class.
class MyClass<T> {
    private T val;

    // A constructor that takes an argument.
    MyClass(T v) { val = v; }
}

```

```
// The default constructor. This constructor
// is NOT used by this program.
MyClass() { val = null; }
// ...

T getVal() { return val; }

// A simple, non-generic class.
class MyClass2 {
    String str;

    // A constructor that takes an argument.
    MyClass2(String s) { str = s; }

    // The default constructor. This
    // constructor is NOT used by this program.
    MyClass2() { str = ""; }

    // ...

    String getVal() { return str; }
}

class ConstructorRefDemo3 {

    // A factory method for class objects. The class must
    // have a constructor that takes one parameter of type T.
    // R specifies the type of object being created.
    static <R,T> R myClassFactory(MyFunc<R, T> cons, T v) {
        return cons.func(v);
    }

    public static void main(String args[])
    {
        // Create a reference to a MyClass constructor.
        // In this case, new refers to the constructor that
        // takes an argument.
        MyFunc<MyClass<Double>, Double> myClassCons = MyClass<Double>::new;

        // Create an instance of MyClass by use of the factory method.
        MyClass<Double> mc = myClassFactory(myClassCons, 100.1);

        // Use the instance of MyClass just created.
        System.out.println("val in mc is " + mc.getVal());

        // Now, create a different class by use of myClassFactory().
        MyFunc<MyClass2, String> myClassCons2 = MyClass2::new;

        // Create an instance of MyClass2 by use of the factory method.
        MyClass2 mc2 = myClassFactory(myClassCons2, "Lambda");
    }
}
```

```

    // Use the instance of MyClass just created.
    System.out.println("str in mc2 is " + mc2.getVal());
}
}

```

The output is shown here:

```

val in mc is 100.1
str in mc2 is Lambda

```

As you can see, **myClassFactory()** is used to create objects of type **MyClass<Double>** and **MyClass2**. Although both classes differ, for example **MyClass** is generic and **MyClass2** is not, both can be created by **myClassFactory()** because they both have constructors that are compatible with **func()** in **MyFunc**. This works because **myClassFactory()** is passed the constructor for the object that it builds. You might want to experiment with this program a bit, trying different classes that you create. Also try creating instances of different types of **MyClass** objects. As you will see, **myClassFactory()** can create any type of object whose class has a constructor that is compatible with **func()** in **MyFunc**. Although this example is quite simple, it hints at the power that constructor references bring to Java.

Before moving on, it is important to mention a second form of the constructor reference syntax that is used for arrays. To create a constructor reference for an array, use this construct:

type[]::new

Here, *type* specifies the type of object being created. For example, assuming the form of **MyClass** as shown in the first constructor reference example (**ConstructorRefDemo**) and given the **MyArrayCreator** interface shown here:

```

interface MyArrayCreator<T> {
    T func(int n);
}

```

the following creates a two-element array of **MyClass** objects and gives each element an initial value:

```

MyArrayCreator<MyClass[]> mcArrayCons = MyClass[] ::new;
MyClass[] a = mcArrayCons.func(2);
a[0] = new MyClass(1);
a[1] = new MyClass(2);

```

Here, the call to **func(2)** causes a two-element array to be created. In general, a functional interface must contain a method that takes a single **int** parameter if it is to be used to refer to an array constructor.

Predefined Functional Interfaces

Up to this point, the examples in this chapter have defined their own functional interfaces so that the fundamental concepts behind lambda expressions and functional interfaces could be clearly illustrated. However, in many cases, you won't need to define your own functional interface because JDK 8 adds a new package called **java.util.function** that

provides several predefined ones. Although we will look at them more closely in Part II, here is a sampling:

Interface	Purpose
UnaryOperator<T>	Apply a unary operation to an object of type T and return the result, which is also of type T . Its method is called apply() .
BinaryOperator<T>	Apply an operation to two objects of type T and return the result, which is also of type T . Its method is called apply() .
Consumer<T>	Apply an operation on an object of type T . Its method is called accept() .
Supplier<T>	Return an object of type T . Its method is called get() .
Function<T, R>	Apply an operation to an object of type T and return the result as an object of type R . Its method is called apply() .
Predicate<T>	Determine if an object of type T fulfills some constraint. Return a boolean value that indicates the outcome. Its method is called test() .

The following program shows the **Function** interface in action by using it to rework the earlier example called **BlockLambdaDemo** that demonstrated block lambdas by implementing a factorial example. That example created its own functional interface called **NumericFunc**, but the built-in **Function** interface could have been used, as this version of the program illustrates:

```
// Use the Function built-in functional interface.

// Import the Function interface.
import java.util.function.Function;

class UseFunctionInterfaceDemo {
    public static void main(String args[])
    {

        // This block lambda computes the factorial of an int value.
        // This time, Function is the functional interface.
        Function<Integer, Integer> factorial = (n) -> {
            int result = 1;
            for(int i=1; i <= n; i++)
                result = i * result;
            return result;
        };

        System.out.println("The factorial of 3 is " + factorial.apply(3));
        System.out.println("The factorial of 5 is " + factorial.apply(5));
    }
}
```

It produces the same output as previous versions of the program.

This page has been intentionally left blank

PART

II

The Java Library

CHAPTER 16

String Handling

CHAPTER 17

Exploring java.lang

CHAPTER 18

java.util Part 1: The Collections Framework

CHAPTER 19

java.util Part 2: More Utility Classes

CHAPTER 20

Input/Output: Exploring java.io

CHAPTER 21

Exploring NIO

CHAPTER 22

Networking

CHAPTER 23

The Applet Class

CHAPTER 24

Event Handling

CHAPTER 25

Introducing the AWT:
Working with Windows,
Graphics, and Text

CHAPTER 26

Using AWT Controls, Layout Managers, and Menus

CHAPTER 27

Images

CHAPTER 28

The Concurrency Utilities

CHAPTER 29

The Stream API

CHAPTER 30

Regular Expressions and
Other Packages

CHAPTER

16

String Handling

A brief overview of Java's string handling was presented in Chapter 7. In this chapter, it is described in detail. As is the case in most other programming languages, in Java a string is a sequence of characters. But, unlike some other languages that implement strings as character arrays, Java implements strings as objects of type **String**.

Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. Also, **String** objects can be constructed a number of ways, making it easy to obtain a string when needed.

Somewhat unexpectedly, when you create a **String** object, you are creating a string that cannot be changed. That is, once a **String** object has been created, you cannot change the characters that comprise that string. At first, this may seem to be a serious restriction. However, such is not the case. You can still perform all types of string operations. The difference is that each time you need an altered version of an existing string, a new **String** object is created that contains the modifications. The original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is desired, Java provides two options: **StringBuffer** and **StringBuilder**. Both hold strings that can be modified after they are created.

The **String**, **StringBuffer**, and **StringBuilder** classes are defined in **java.lang**. Thus, they are available to all programs automatically. All are declared **final**, which means that none of these classes may be subclassed. This allows certain optimizations that increase performance to take place on common string operations. All three implement the **CharSequence** interface.

One last point: To say that the strings within objects of type **String** are unchangeable means that the contents of the **String** instance cannot be changed after it has been created.

However, a variable declared as a **String** reference can be changed to point at some other **String** object at any time.

The String Constructors

The **String** class supports several constructors. To create an empty **String**, call the default constructor. For example,

```
String s = new String();
```

will create an instance of **String** with no characters in it.

Frequently, you will want to create strings that have initial values. The **String** class provides a variety of constructors to handle this. To create a **String** initialized by an array of characters, use the constructor shown here:

```
String(char chars[ ])
```

Here is an example:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

This constructor initializes **s** with the string "abc".

You can specify a subrange of a character array as an initializer using the following constructor:

```
String(char chars[ ], int startIndex, int numChars)
```

Here, *startIndex* specifies the index at which the subrange begins, and *numChars* specifies the number of characters to use. Here is an example:

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
```

This initializes **s** with the characters **cde**.

You can construct a **String** object that contains the same character sequence as another **String** object using this constructor:

```
String(String strObj)
```

Here, *strObj* is a **String** object. Consider this example:

```
// Construct one String from another.
class MakeString {
    public static void main(String args[]) {
        char c[] = {'J', 'a', 'v', 'a'};
        String s1 = new String(c);
        String s2 = new String(s1);

        System.out.println(s1);
        System.out.println(s2);
    }
}
```

The output from this program is as follows:

```
Java  
Java
```

As you can see, **s1** and **s2** contain the same string.

Even though Java's **char** type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set. Because 8-bit ASCII strings are common, the **String** class provides constructors that initialize a string when given a **byte** array. Two forms are shown here:

```
String(byte chrs[ ])  
String(byte chrs[ ], int startIndex, int numChars)
```

Here, *chrs* specifies the array of bytes. The second form allows you to specify a subrange. In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform. The following program illustrates these constructors:

```
// Construct string from subset of char array.  
class SubStringCons {  
    public static void main(String args[]) {  
        byte ascii[] = {65, 66, 67, 68, 69, 70};  
  
        String s1 = new String(ascii);  
        System.out.println(s1);  
  
        String s2 = new String(ascii, 2, 3);  
        System.out.println(s2);  
    }  
}
```

This program generates the following output:

```
ABCDEF  
CDE
```

Extended versions of the byte-to-string constructors are also defined in which you can specify the character encoding that determines how bytes are converted to characters. However, you will often want to use the default encoding provided by the platform.

NOTE The contents of the array are copied whenever you create a **String** object from an array. If you modify the contents of the array after you have created the string, the **String** will be unchanged.

You can construct a **String** from a **StringBuffer** by using the constructor shown here:

```
String(StringBuffer strBufObj)
```

You can construct a **String** from a **StringBuilder** by using this constructor:

```
String(StringBuilder strBuildObj)
```

The following constructor supports the extended Unicode character set:

```
String(int codePoints[ ], int startIndex, int numChars)
```

Here, *codePoints* is an array that contains Unicode code points. The resulting string is constructed from the range that begins at *startIndex* and runs for *numChars*.

There are also constructors that let you specify a **Charset**.

NOTE A discussion of Unicode code points and how they are handled by Java is found in Chapter 17.

String Length

The length of a string is the number of characters that it contains. To obtain this value, call the **length()** method, shown here:

```
int length()
```

The following fragment prints "3", since there are three characters in the string **s**:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

Special String Operations

Because strings are a common and important part of programming, Java has added special support for several string operations within the syntax of the language. These operations include the automatic creation of new **String** instances from string literals, concatenation of multiple **String** objects by use of the + operator, and the conversion of other data types to a string representation. There are explicit methods available to perform all of these functions, but Java does them automatically as a convenience for the programmer and to add clarity.

String Literals

The earlier examples showed how to explicitly create a **String** instance from an array of characters by using the **new** operator. However, there is an easier way to do this using a string literal. For each string literal in your program, Java automatically constructs a **String** object. Thus, you can use a string literal to initialize a **String** object. For example, the following code fragment creates two equivalent strings:

```
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);

String s2 = "abc"; // use string literal
```

Because a **String** object is created for every string literal, you can use a string literal anywhere you can use a **String** object. For example, you can call methods directly on a quoted string as if it were an object reference, as the following statement shows. It calls the **length()** method on the string "abc". As expected, it prints "3".

```
System.out.println("abc".length());
```

String Concatenation

In general, Java does not allow operators to be applied to **String** objects. The one exception to this rule is the + operator, which concatenates two strings, producing a **String** object as the result. This allows you to chain together a series of + operations. For example, the following fragment concatenates three strings:

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

This displays the string "He is 9 years old."

One practical use of string concatenation is found when you are creating very long strings. Instead of letting long strings wrap around within your source code, you can break them into smaller pieces, using the + to concatenate them. Here is an example:

```
// Using concatenation to prevent long lines.
class ConCat {
    public static void main(String args[]) {
        String longStr = "This could have been " +
            "a very long line that would have " +
            "wrapped around. But string concatenation " +
            "prevents this.";

        System.out.println(longStr);
    }
}
```

String Concatenation with Other Data Types

You can concatenate strings with other types of data. For example, consider this slightly different version of the earlier example:

```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);
```

In this case, **age** is an **int** rather than another **String**, but the output produced is the same as before. This is because the **int** value in **age** is automatically converted into its string representation within a **String** object. This string is then concatenated as before. The compiler will convert an operand to its string equivalent whenever the other operand of the + is an instance of **String**.

Be careful when you mix other types of operations with string concatenation expressions, however. You might get surprising results. Consider the following:

```
String s = "four: " + 2 + 2;
System.out.println(s);
```

This fragment displays

```
four: 22
```

rather than the

```
four: 4
```

that you probably expected. Here's why. Operator precedence causes the concatenation of "four" with the string equivalent of 2 to take place first. This result is then concatenated with the string equivalent of 2 a second time. To complete the integer addition first, you must use parentheses, like this:

```
String s = "four: " + (2 + 2);
```

Now **s** contains the string "four: 4".

String Conversion and **toString()**

When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method **valueOf()** defined by **String**. **valueOf()** is overloaded for all the primitive types and for type **Object**. For the primitive types, **valueOf()** returns a string that contains the human-readable equivalent of the value with which it is called. For objects, **valueOf()** calls the **toString()** method on the object. We will look more closely at **valueOf()** later in this chapter. Here, let's examine the **toString()** method, because it is the means by which you can determine the string representation for objects of classes that you create.

Every class implements **toString()** because it is defined by **Object**. However, the default implementation of **toString()** is seldom sufficient. For most important classes that you create, you will want to override **toString()** and provide your own string representations. Fortunately, this is easy to do. The **toString()** method has this general form:

```
String toString()
```

To implement **toString()**, simply return a **String** object that contains the human-readable string that appropriately describes an object of your class.

By overriding **toString()** for classes that you create, you allow them to be fully integrated into Java's programming environment. For example, they can be used in **print()** and **println()** statements and in concatenation expressions. The following program demonstrates this by overriding **toString()** for the **Box** class:

```
// Override toString() for Box class.
class Box {
    double width;
    double height;
    double depth;

    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    public String toString() {
        return "Dimensions are " + width + " by " +
               height + " with a depth of " + depth;
    }
}
```

```

        depth + " by " + height + ".";
    }
}

class toStringDemo {
    public static void main(String args[]) {
        Box b = new Box(10, 12, 14);
        String s = "Box b: " + b; // concatenate Box object

        System.out.println(b); // convert Box to string
        System.out.println(s);
    }
}

```

The output of this program is shown here:

```

Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0

```

As you can see, **Box**'s **toString()** method is automatically invoked when a **Box** object is used in a concatenation expression or in a call to **println()**.

Character Extraction

The **String** class provides a number of ways in which characters can be extracted from a **String** object. Several are examined here. Although the characters that comprise a string within a **String** object cannot be indexed as if they were a character array, many of the **String** methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

charAt()

To extract a single character from a **String**, you can refer directly to an individual character via the **charAt()** method. It has this general form:

```
char charAt(int where)
```

Here, *where* is the index of the character that you want to obtain. The value of *where* must be nonnegative and specify a location within the string. **charAt()** returns the character at the specified location. For example,

```
char ch;
ch = "abc".charAt(1);
```

assigns the value **b** to **ch**.

getChars()

If you need to extract more than one character at a time, you can use the **getChars()** method. It has this general form:

```
void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
```

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from *sourceStart* through *sourceEnd*-1. The array that will receive the characters is specified by *target*. The index within *target* at which the substring will be copied is passed in *targetStart*. Care must be taken to assure that the *target* array is large enough to hold the number of characters in the specified substring.

The following program demonstrates **getChars()**:

```
class getCharsDemo {
    public static void main(String args[]) {
        String s = "This is a demo of the getChars method.";
        int start = 10;
        int end = 14;
        char buf[] = new char[end - start];

        s.getChars(start, end, buf, 0);
        System.out.println(buf);
    }
}
```

Here is the output of this program:

```
demo
```

getBytes()

There is an alternative to **getChars()** that stores the characters in an array of bytes. This method is called **getBytes()**, and it uses the default character-to-byte conversions provided by the platform. Here is its simplest form:

```
byte[ ] getBytes()
```

Other forms of **getBytes()** are also available. **getBytes()** is most useful when you are exporting a **String** value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

toCharArray()

If you want to convert all the characters in a **String** object into a character array, the easiest way is to call **toCharArray()**. It returns an array of characters for the entire string. It has this general form:

```
char[ ] toCharArray()
```

This function is provided as a convenience, since it is possible to use **getChars()** to achieve the same result.

String Comparison

The **String** class includes a number of methods that compare strings or substrings within strings. Several are examined here.

equals() and equalsIgnoreCase()

To compare two strings for equality, use **equals()**. It has this general form:

boolean equals(Object str)

Here, `str` is the **String** object being compared with the invoking **String** object. It returns **true** if the strings contain the same characters in the same order, and **false** otherwise. The comparison is case-sensitive.

To perform a comparison that ignores case differences, call `equalsIgnoreCase()`. When it compares two strings, it considers **A-Z** to be the same as **a-z**. It has this general form:

boolean equalsIgnoreCase(String str)

Here, `str` is the **String** object being compared with the invoking **String** object. It, too, returns **true** if the strings contain the same characters in the same order, and **false** otherwise.

Here is an example that demonstrates `equals()` and `equalsIgnoreCase()`:

```
// Demonstrate equals() and equalsIgnoreCase().  
class equalsDemo {  
    public static void main(String args[]) {  
        String s1 = "Hello";  
        String s2 = "Hello";  
        String s3 = "Good-bye";  
        String s4 = "HELLO";  
        System.out.println(s1 + " equals " + s2 + " -> " +  
                           s1.equals(s2));  
        System.out.println(s1 + " equals " + s3 + " -> " +  
                           s1.equals(s3));  
        System.out.println(s1 + " equals " + s4 + " -> " +  
                           s1.equals(s4));  
        System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +  
                           s1.equalsIgnoreCase(s4));  
    }  
}
```

The output from the program is shown here:

```
Hello.equals(Hello) -> true  
Hello.equals(Good-bye) -> false  
Hello.equals(HELLO) -> false  
Hello.equalsIgnoreCase(HELLO) -> true
```

regionMatches()

The **regionMatches()** method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows you to ignore case in such comparisons. Here are the general forms for these two methods:

```
boolean regionMatches(int startIndex, String str2,  
                     int str2startIndex, int numChars)
```

```
boolean regionMatches(boolean ignoreCase,
                     int startIndex, String str2,
                     int str2StartIndex, int numChars)
```

For both versions, *startIndex* specifies the index at which the region begins within the invoking **String** object. The **String** being compared is specified by *str2*. The index at which the comparison will start within *str2* is specified by *str2startIndex*. The length of the substring being compared is passed in *numChars*. In the second version, if *ignoreCase* is **true**, the case of the characters is ignored. Otherwise, case is significant.

startsWith() and endsWith()

String defines two methods that are, more or less, specialized forms of **regionMatches()**. The **startsWith()** method determines whether a given **String** begins with a specified string. Conversely, **endsWith()** determines whether the **String** in question ends with a specified string. They have the following general forms:

```
boolean startsWith(String str)
boolean endsWith(String str)
```

Here, *str* is the **String** being tested. If the string matches, **true** is returned. Otherwise, **false** is returned. For example,

```
"Foobar".endsWith("bar")
```

and

```
"Foobar".startsWith("Foo")
```

are both **true**.

A second form of **startsWith()**, shown here, lets you specify a starting point:

```
boolean startsWith(String str, int startIndex)
```

Here, *startIndex* specifies the index into the invoking string at which point the search will begin. For example,

```
"Foobar".startsWith("bar", 3)
```

returns **true**.

equals() Versus ==

It is important to understand that the **equals()** method and the **==** operator perform two different operations. As just explained, the **equals()** method compares the characters inside a **String** object. The **==** operator compares two object references to see whether they refer to the same instance. The following program shows how two different **String** objects can contain the same characters, but references to these objects will not compare as equal:

```
// equals() vs ==
class EqualsNotEqualTo {
    public static void main(String args[]) {
```

```

String s1 = "Hello";
String s2 = new String(s1);

System.out.println(s1 + " equals " + s2 + " -> " +
                   s1.equals(s2));
System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
}
}

```

The variable **s1** refers to the **String** instance created by "Hello". The object referred to by **s2** is created with **s1** as an initializer. Thus, the contents of the two **String** objects are identical, but they are distinct objects. This means that **s1** and **s2** do not refer to the same objects and are, therefore, not `==`, as is shown here by the output of the preceding example:

```
Hello equals Hello -> true
Hello == Hello -> false
```

compareTo()

Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is *less than*, *equal to*, or *greater than* the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The method **compareTo()** serves this purpose. It is specified by the **Comparable<T>** interface, which **String** implements. It has this general form:

```
int compareTo(String str)
```

Here, *str* is the **String** being compared with the invoking **String**. The result of the comparison is returned and is interpreted as shown here:

Value	Meaning
Less than zero	The invoking string is less than <i>str</i> .
Greater than zero	The invoking string is greater than <i>str</i> .
Zero	The two strings are equal.

Here is a sample program that sorts an array of strings. The program uses **compareTo()** to determine sort ordering for a bubble sort:

```

// A bubble sort for Strings.
class SortString {
    static String arr[] = {
        "Now", "is", "the", "time", "for", "all", "good", "men",
        "to", "come", "to", "the", "aid", "of", "their", "country"
    };
    public static void main(String args[]) {
        for(int j = 0; j < arr.length; j++) {
            for(int i = j + 1; i < arr.length; i++) {
                if(arr[i].compareTo(arr[j]) < 0) {
                    String t = arr[j];

```

```
        arr[j] = arr[i];
        arr[i] = t;
    }
}
System.out.println(arr[j]);
}
}
```

The output of this program is the list of words:

Now
aid
all
come
country
for
good
is
men
of
the
the
their
time
to
to

As you can see from the output of this example, `compareTo()` takes into account uppercase and lowercase letters. The word "Now" came out before all the others because it begins with an uppercase letter, which means it has a lower value in the ASCII character set.

If you want to ignore case differences when comparing two strings, use `compareToIgnoreCase()`, as shown here:

```
int compareToIgnoreCase(String str)
```

This method returns the same results as `compareTo()`, except that case differences are ignored. You might want to try substituting it into the previous program. After doing so, "Now" will no longer be first.

Searching Strings

The **String** class provides two methods that allow you to search a string for a specified character or substring:

- **indexOf()** Searches for the first occurrence of a character or substring.
 - **lastIndexOf()** Searches for the last occurrence of a character or substring.

These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or -1 on failure.

To search for the first occurrence of a character, use

int indexOf(int ch)

To search for the last occurrence of a character, use

```
int lastIndexOf(int ch)
```

Here, *ch* is the character being sought.

To search for the first or last occurrence of a substring, use

```
int indexOf(String str)  
int lastIndexOf(String str)
```

Here, *str* specifies the substring.

You can specify a starting point for the search using these forms:

```
int indexOf(int ch, int startIndex)  
int lastIndexOf(int ch, int startIndex)  
  
int indexOf(String str, int startIndex)  
int lastIndexOf(String str, int startIndex)
```

Here, *startIndex* specifies the index at which point the search begins. For **indexOf()**, the search runs from *startIndex* to the end of the string. For **lastIndexOf()**, the search runs from *startIndex* to zero.

The following example shows how to use the various index methods to search inside of a **String**:

```
// Demonstrate indexOf() and lastIndexOf().  
class indexOfDemo {  
    public static void main(String args[]) {  
        String s = "Now is the time for all good men " +  
                  "to come to the aid of their country.";  
  
        System.out.println(s);  
        System.out.println("indexOf(t) = " +  
                           s.indexOf('t'));  
        System.out.println("lastIndexOf(t) = " +  
                           s.lastIndexOf('t'));  
        System.out.println("indexOf(the) = " +  
                           s.indexOf("the"));  
        System.out.println("lastIndexOf(the) = " +  
                           s.lastIndexOf("the"));  
        System.out.println("indexOf(t, 10) = " +  
                           s.indexOf('t', 10));  
        System.out.println("lastIndexOf(t, 60) = " +  
                           s.lastIndexOf('t', 60));  
        System.out.println("indexOf(the, 10) = " +  
                           s.indexOf("the", 10));  
        System.out.println("lastIndexOf(the, 60) = " +  
                           s.lastIndexOf("the", 60));  
    }  
}
```

Here is the output of this program:

```
Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```

Modifying a String

Because **String** objects are immutable, whenever you want to modify a **String**, you must either copy it into a **StringBuffer** or **StringBuilder**, or use a **String** method that constructs a new copy of the string with your modifications complete. A sampling of these methods are described here.

substring()

You can extract a substring using **substring()**. It has two forms. The first is

```
String substring(int startIndex)
```

Here, *startIndex* specifies the index at which the substring will begin. This form returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string.

The second form of **substring()** allows you to specify both the beginning and ending index of the substring:

```
String substring(int startIndex, int endIndex)
```

Here, *startIndex* specifies the beginning index, and *endIndex* specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

The following program uses **substring()** to replace all instances of one substring with another within a string:

```
// Substring replacement.
class StringReplace {
    public static void main(String args[]) {
        String org = "This is a test. This is, too.";
        String search = "is";
        String sub = "was";
        String result = "";
        int i;

        do { // replace all matching substrings
            System.out.println(org);
            i = org.indexOf(search);
            if(i != -1) {
                result = org.substring(0, i);
                result = result + sub;
            }
        }
    }
}
```

```

        result = result + org.substring(i + search.length());
        org = result;
    }
} while(i != -1);
}
}

```

The output from this program is shown here:

```

This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.

```

concat()

You can concatenate two strings using **concat()**, shown here:

```
String concat(String str)
```

This method creates a new object that contains the invoking string with the contents of *str* appended to the end. **concat()** performs the same function as **+**. For example,

```
String s1 = "one";
String s2 = s1.concat("two");
```

puts the string "onetwo" into **s2**. It generates the same result as the following sequence:

```
String s1 = "one";
String s2 = s1 + "two";
```

replace()

The **replace()** method has two forms. The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:

```
String replace(char original, char replacement)
```

Here, *original* specifies the character to be replaced by the character specified by *replacement*. The resulting string is returned. For example,

```
String s = "Hello".replace('l', 'w');
```

puts the string "Hewwo" into **s**.

The second form of **replace()** replaces one character sequence with another. It has this general form:

```
String replace(CharSequence original, CharSequence replacement)
```

trim()

The **trim()** method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:

```
String trim()
```

Here is an example:

```
String s = "Hello World".trim();
```

This puts the string "Hello World" into **s**.

The **trim()** method is quite useful when you process user commands. For example, the following program prompts the user for the name of a state and then displays that state's capital. It uses **trim()** to remove any leading or trailing whitespace that may have inadvertently been entered by the user.

```
// Using trim() to process commands.
import java.io.*;

class UseTrim {
    public static void main(String args[])
        throws IOException
    {
        // create a BufferedReader using System.in
        BufferedReader br = new
            BufferedReader(new InputStreamReader(System.in));
        String str;

        System.out.println("Enter 'stop' to quit.");
        System.out.println("Enter State: ");
        do {
            str = br.readLine();
            str = str.trim(); // remove whitespace

            if(str.equals("Illinois"))
                System.out.println("Capital is Springfield.");
            else if(str.equals("Missouri"))
                System.out.println("Capital is Jefferson City.");
            else if(str.equals("California"))
                System.out.println("Capital is Sacramento.");
            else if(str.equals("Washington"))
                System.out.println("Capital is Olympia.");
            // ...
        } while(!str.equals("stop"));
    }
}
```

Data Conversion Using valueOf()

The **valueOf()** method converts data from its internal format into a human-readable form. It is a static method that is overloaded within **String** for all of Java's built-in types so that each type can be converted properly into a string. **valueOf()** is also overloaded for type

Object, so an object of any class type you create can also be used as an argument. (Recall that **Object** is a superclass for all classes.) Here are a few of its forms:

```
static String valueOf(double num)
static String valueOf(long num)
static String valueOf(Object ob)
static String valueOf(char chars[ ])
```

As discussed earlier, **valueOf()** is called when a string representation of some other type of data is needed—for example, during concatenation operations. You can call this method directly with any data type and get a reasonable **String** representation. All of the simple types are converted to their common **String** representation. Any object that you pass to **valueOf()** will return the result of a call to the object's **toString()** method. In fact, you could just call **toString()** directly and get the same result.

For most arrays, **valueOf()** returns a rather cryptic string, which indicates that it is an array of some type. For arrays of **char**, however, a **String** object is created that contains the characters in the **char** array. There is a special version of **valueOf()** that allows you to specify a subset of a **char** array. It has this general form:

```
static String valueOf(char chars[ ], int startIndex, int numChars)
```

Here, *chars* is the array that holds the characters, *startIndex* is the index into the array of characters at which the desired substring begins, and *numChars* specifies the length of the substring.

Changing the Case of Characters Within a String

The method **toLowerCase()** converts all the characters in a string from uppercase to lowercase. The **toUpperCase()** method converts all the characters in a string from lowercase to uppercase. Nonalphabetical characters, such as digits, are unaffected. Here are the simplest forms of these methods:

```
String toLowerCase()
String toUpperCase()
```

Both methods return a **String** object that contains the uppercase or lowercase equivalent of the invoking **String**. The default locale governs the conversion in both cases.

Here is an example that uses **toLowerCase()** and **toUpperCase()**:

```
// Demonstrate toUpperCase() and toLowerCase().

class ChangeCase {
    public static void main(String args[])
    {
        String s = "This is a test.";
        System.out.println("Original: " + s);

        String upper = s.toUpperCase();
        String lower = s.toLowerCase();
```

```

        System.out.println("Uppercase: " + upper);
        System.out.println("Lowercase: " + lower);
    }
}

```

The output produced by the program is shown here:

```

Original: This is a test.
Uppercase: THIS IS A TEST.
Lowercase: this is a test.

```

One other point: Overloaded versions of `toLowerCase()` and `toUpperCase()` that let you specify a `Locale` object to govern the conversion are also supplied. Specifying the locale can be quite important in some cases and can help internationalize your application.

Joining Strings

JDK 8 adds a new method to `String` called `join()`. It is used to concatenate two or more strings, separating each string with a delimiter, such as a space or a comma. It has two forms. Its first is shown here:

```
static String join(CharSequence delim, CharSequence ... strs)
```

Here, `delim` specifies the delimiter used to separate the character sequences specified by `strs`. Because `String` implements the `CharSequence` interface, `strs` can be a list of strings. (See Chapter 17 for information on `CharSequence`.) The following program demonstrates this version of `join()`:

```

// Demonstrate the join() method defined by String.
class StringJoinDemo {
    public static void main(String args[]) {

        String result = String.join(" ", "Alpha", "Beta", "Gamma");
        System.out.println(result);

        result = String.join(", ", "John", "ID#: 569",
                             "E-mail: John@HerbSchildt.com");
        System.out.println(result);
    }
}

```

The output is shown here:

```

Alpha Beta Gamma
John, ID#: 569, E-mail: John@HerbSchildt.com

```

In the first call to `join()`, a space is inserted between each string. In the second call, the delimiter is a comma followed by a space. This illustrates that the delimiter need not be just a single character.

The second form of `join()` lets you join a list of strings obtained from an object that implements the **Iterable** interface. **Iterable** is implemented by the Collections Framework classes described in Chapter 18, among others. See Chapter 17 for information on **Iterable**.

Additional String Methods

In addition to those methods discussed earlier, **String** has many other methods, including those summarized in the following table:

Method	Description
<code>int codePointAt(int <i>i</i>)</code>	Returns the Unicode code point at the location specified by <i>i</i> .
<code>int codePointBefore(int <i>i</i>)</code>	Returns the Unicode code point at the location that precedes that specified by <i>i</i> .
<code>int codePointCount(int <i>start</i>, int <i>end</i>)</code>	Returns the number of code points in the portion of the invoking String that are between <i>start</i> and <i>end</i> –1.
<code>boolean contains(CharSequence <i>str</i>)</code>	Returns true if the invoking object contains the string specified by <i>str</i> . Returns false otherwise.
<code>boolean contentEquals(CharSequence <i>str</i>)</code>	Returns true if the invoking string contains the same string as <i>str</i> . Otherwise, returns false .
<code>boolean contentEquals(StringBuffer <i>str</i>)</code>	Returns true if the invoking string contains the same string as <i>str</i> . Otherwise, returns false .
<code>static String format(String <i>fmtstr</i>, Object ... <i>args</i>)</code>	Returns a string formatted as specified by <i>fmtstr</i> . (See Chapter 19 for details on formatting.)
<code>static String format(Locale <i>loc</i>, String <i>fmtstr</i>, Object ... <i>args</i>)</code>	Returns a string formatted as specified by <i>fmtstr</i> . Formatting is governed by the locale specified by <i>loc</i> . (See Chapter 19 for details on formatting.)
<code>boolean isEmpty()</code>	Returns true if the invoking string contains no characters and has a length of zero.
<code>boolean matches(string <i>regExp</i>)</code>	Returns true if the invoking string matches the regular expression passed in <i>regExp</i> . Otherwise, returns false .
<code>int offsetByCodePoints(int <i>start</i>, int <i>num</i>)</code>	Returns the index within the invoking string that is <i>num</i> code points beyond the starting index specified by <i>start</i> .
<code>String replaceFirst(String <i>regExp</i>, String <i>newStr</i>)</code>	Returns a string in which the first substring that matches the regular expression specified by <i>regExp</i> is replaced by <i>newStr</i> .
<code>String replaceAll(String <i>regExp</i>, String <i>newStr</i>)</code>	Returns a string in which all substrings that match the regular expression specified by <i>regExp</i> are replaced by <i>newStr</i> .
<code>String[] split(String <i>regExp</i>)</code>	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in <i>regExp</i> .

Method	Description
<code>String[] split(String <i>regExp</i>, int <i>max</i>)</code>	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in <i>regExp</i> . The number of pieces is specified by <i>max</i> . If <i>max</i> is negative, then the invoking string is fully decomposed. Otherwise, if <i>max</i> contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If <i>max</i> is zero, the invoking string is fully decomposed, but no trailing empty strings will be included.
<code>CharSequence subSequence(int <i>startIndex</i>, int <i>stopIndex</i>)</code>	Returns a substring of the invoking string, beginning at <i>startIndex</i> and stopping at <i>stopIndex</i> . This method is required by the CharSequence interface, which is implemented by String .

Notice that several of these methods work with regular expressions. Regular expressions are described in Chapter 30.

StringBuffer

StringBuffer supports a modifiable string. As you know, **String** represents fixed-length, immutable character sequences. In contrast, **StringBuffer** represents growable and writable character sequences. **StringBuffer** may have characters and substrings inserted in the middle or appended to the end. **StringBuffer** will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

StringBuffer Constructors

StringBuffer defines these four constructors:

```
StringBuffer()
StringBuffer(int size)
StringBuffer(String str)
StringBuffer(CharSequence chars)
```

The default constructor (the one with no parameters) reserves room for 16 characters without reallocation. The second version accepts an integer argument that explicitly sets the size of the buffer. The third version accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more characters without reallocation. **StringBuffer** allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, **StringBuffer** reduces the number of reallocations that take place. The fourth constructor creates an object that contains the character sequence contained in *chars* and reserves room for 16 more characters.

length() and capacity()

The current length of a **StringBuffer** can be found via the **length()** method, while the total allocated capacity can be found through the **capacity()** method. They have the following general forms:

```
int length()
int capacity()
```

Here is an example:

```
// StringBuffer length vs. capacity.
class StringBufferDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");

        System.out.println("buffer = " + sb);
        System.out.println("length = " + sb.length());
        System.out.println("capacity = " + sb.capacity());
    }
}
```

Here is the output of this program, which shows how **StringBuffer** reserves extra space for additional manipulations:

```
buffer = Hello
length = 5
capacity = 21
```

Since **sb** is initialized with the string "Hello" when it is created, its length is 5. Its capacity is 21 because room for 16 additional characters is automatically added.

ensureCapacity()

If you want to preallocate room for a certain number of characters after a **StringBuffer** has been constructed, you can use **ensureCapacity()** to set the size of the buffer. This is useful if you know in advance that you will be appending a large number of small strings to a **StringBuffer**. **ensureCapacity()** has this general form:

```
void ensureCapacity(int minCapacity)
```

Here, *minCapacity* specifies the minimum size of the buffer. (A buffer larger than *minCapacity* may be allocated for reasons of efficiency.)

setLength()

To set the length of the string within a **StringBuffer** object, use **setLength()**. Its general form is shown here:

```
void setLength(int len)
```

Here, *len* specifies the length of the string. This value must be nonnegative.

When you increase the size of the string, null characters are added to the end. If you call **setLength()** with a value less than the current value returned by **length()**, then the

characters stored beyond the new length will be lost. The **setCharAtDemo** sample program in the following section uses **setLength()** to shorten a **StringBuffer**.

charAt() and setCharAt()

The value of a single character can be obtained from a **StringBuffer** via the **charAt()** method. You can set the value of a character within a **StringBuffer** using **setCharAt()**. Their general forms are shown here:

```
char charAt(int where)
void setCharAt(int where, char ch)
```

For **charAt()**, *where* specifies the index of the character being obtained. For **setCharAt()**, *where* specifies the index of the character being set, and *ch* specifies the new value of that character. For both methods, *where* must be nonnegative and must not specify a location beyond the end of the string.

The following example demonstrates **charAt()** and **setCharAt()**:

```
// Demonstrate charAt() and setCharAt().
class setCharAtDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");
        System.out.println("buffer before = " + sb);
        System.out.println("charAt(1) before = " + sb.charAt(1));

        sb.setCharAt(1, 'i');
        sb.setLength(2);
        System.out.println("buffer after = " + sb);
        System.out.println("charAt(1) after = " + sb.charAt(1));
    }
}
```

Here is the output generated by this program:

```
buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i
```

getChars()

To copy a substring of a **StringBuffer** into an array, use the **getChars()** method. It has this general form:

```
void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
```

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. This means that the substring contains the characters from *sourceStart* through *sourceEnd*-1. The array that will receive the characters is specified by *target*. The index within *target* at which the substring will be copied is passed in *targetStart*. Care must be taken to assure that the *target* array is large enough to hold the number of characters in the specified substring.

append()

The **append()** method concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object. It has several overloaded versions. Here are a few of its forms:

```
StringBuffer append(String str)
StringBuffer append(int num)
StringBuffer append(Object obj)
```

The string representation of each parameter is obtained, often by calling **String.valueOf()**. The result is appended to the current **StringBuffer** object. The buffer itself is returned by each version of **append()**. This allows subsequent calls to be chained together, as shown in the following example:

```
// Demonstrate append().
class appendDemo {
    public static void main(String args[]) {
        String s;
        int a = 42;
        StringBuffer sb = new StringBuffer(40);

        s = sb.append("a = ").append(a).append("!").toString();
        System.out.println(s);
    }
}
```

The output of this example is shown here:

```
a = 42!
```

insert()

The **insert()** method inserts one string into another. It is overloaded to accept values of all the primitive types, plus **Strings**, **Objects**, and **CharSequences**. Like **append()**, it obtains the string representation of the value it is called with. This string is then inserted into the invoking **StringBuffer** object. These are a few of its forms:

```
StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object obj)
```

Here, *index* specifies the index at which point the string will be inserted into the invoking **StringBuffer** object.

The following sample program inserts "like" between "I" and "Java":

```
// Demonstrate insert().
class insertDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("I Java!");
    }
}
```

```

        sb.insert(2, "like ");
        System.out.println(sb);
    }
}

```

The output of this example is shown here:

```
I like Java!
```

reverse()

You can reverse the characters within a **StringBuffer** object using **reverse()**, shown here:

```
StringBuffer reverse()
```

This method returns the reverse of the object on which it was called. The following program demonstrates **reverse()**:

```

// Using reverse() to reverse a StringBuffer.
class ReverseDemo {
    public static void main(String args[]) {
        StringBuffer s = new StringBuffer("abcdef");

        System.out.println(s);
        s.reverse();
        System.out.println(s);
    }
}

```

Here is the output produced by the program:

```
abcdef
fedcba
```

delete() and deleteCharAt()

You can delete characters within a **StringBuffer** by using the methods **delete()** and **deleteCharAt()**. These methods are shown here:

```
StringBuffer delete(int startIndex, int endIndex)
StringBuffer deleteCharAt(int loc)
```

The **delete()** method deletes a sequence of characters from the invoking object. Here, *startIndex* specifies the index of the first character to remove, and *endIndex* specifies an index one past the last character to remove. Thus, the substring deleted runs from *startIndex* to *endIndex*-1. The resulting **StringBuffer** object is returned.

The **deleteCharAt()** method deletes the character at the index specified by *loc*. It returns the resulting **StringBuffer** object.

Here is a program that demonstrates the **delete()** and **deleteCharAt()** methods:

```

// Demonstrate delete() and deleteCharAt()
class deleteDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("This is a test.");

```

```

        sb.delete(4, 7);
        System.out.println("After delete: " + sb);

        sb.deleteCharAt(0);
        System.out.println("After deleteCharAt: " + sb);
    }
}

```

The following output is produced:

```

After delete: This a test.
After deleteCharAt: his a test.

```

replace()

You can replace one set of characters with another set inside a **StringBuffer** object by calling **replace()**. Its signature is shown here:

```
StringBuffer replace(int startIndex, int endIndex, String str)
```

The substring being replaced is specified by the indexes *startIndex* and *endIndex*. Thus, the substring at *startIndex* through *endIndex*-1 is replaced. The replacement string is passed in *str*. The resulting **StringBuffer** object is returned.

The following program demonstrates **replace()**:

```

// Demonstrate replace()
class replaceDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("This is a test.");

        sb.replace(5, 7, "was");
        System.out.println("After replace: " + sb);
    }
}

```

Here is the output:

```
After replace: This was a test.
```

substring()

You can obtain a portion of a **StringBuffer** by calling **substring()**. It has the following two forms:

```
String substring(int startIndex)
String substring(int startIndex, int endIndex)
```

The first form returns the substring that starts at *startIndex* and runs to the end of the invoking **StringBuffer** object. The second form returns the substring that starts at *startIndex* and runs through *endIndex*-1. These methods work just like those defined for **String** that were described earlier.

Additional StringBuffer Methods

In addition to those methods just described, **StringBuffer** supplies several others, including those summarized in the following table:

Method	Description
<code>StringBuffer appendCodePoint(int ch)</code>	Appends a Unicode code point to the end of the invoking object. A reference to the object is returned.
<code>int codePointAt(int i)</code>	Returns the Unicode code point at the location specified by <i>i</i> .
<code>int codePointBefore(int i)</code>	Returns the Unicode code point at the location that precedes that specified by <i>i</i> .
<code>int codePointCount(int start, int end)</code>	Returns the number of code points in the portion of the invoking String that are between <i>start</i> and <i>end</i> -1.
<code>int indexOf(String str)</code>	Sets the invoking StringBuffer for the first occurrence of <i>str</i> . Returns the index of the match, or -1 if no match is found.
<code>int indexOf(String str, int startIndex)</code>	Sets the invoking StringBuffer for the first occurrence of <i>str</i> , beginning at <i>startIndex</i> . Returns the index of the match, or -1 if no match is found.
<code>int lastIndexOf(String str)</code>	Sets the invoking StringBuffer for the last occurrence of <i>str</i> . Returns the index of the match, or -1 if no match is found.
<code>int lastIndexOf(String str, int startIndex)</code>	Sets the invoking StringBuffer for the last occurrence of <i>str</i> , beginning at <i>startIndex</i> . Returns the index of the match, or -1 if no match is found.
<code>int offsetByCodePoints(int start, int num)</code>	Returns the index within the invoking string that is <i>num</i> code points beyond the starting index specified by <i>start</i> .
<code>CharSequence subSequence(int startIndex, int stopIndex)</code>	Returns a substring of the invoking string, beginning at <i>startIndex</i> and stopping at <i>stopIndex</i> . This method is required by the CharSequence interface, which is implemented by StringBuffer .
<code>void trimToSize()</code>	Requests that the size of the character buffer for the invoking object be reduced to better fit the current contents.

The following program demonstrates **indexOf()** and **lastIndexOf()**:

```
class IndexOfDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("one two one");
        int i;

        i = sb.indexOf("one");
        System.out.println("First index: " + i);

        i = sb.lastIndexOf("one");
        System.out.println("Last index: " + i);
    }
}
```

The output is shown here:

```
First index: 0  
Last index: 8
```

StringBuilder

Introduced by JDK 5, **StringBuilder** is a relatively recent addition to Java's string handling capabilities. **StringBuilder** is similar to **StringBuffer** except for one important difference: it is not synchronized, which means that it is not thread-safe. The advantage of **StringBuilder** is faster performance. However, in cases in which a mutable string will be accessed by multiple threads, and no external synchronization is employed, you must use **StringBuffer** rather than **StringBuilder**.

This page has been intentionally left blank

CHAPTER

17

Exploring `java.lang`

This chapter discusses those classes and interfaces defined by `java.lang`. As you know, `java.lang` is automatically imported into all programs. It contains classes and interfaces that are fundamental to virtually all of Java programming. It is Java's most widely used package.

`java.lang` includes the following classes:

<code>Boolean</code>	<code>Enum</code>	<code>Process</code>	<code>String</code>
<code>Byte</code>	<code>Float</code>	<code>ProcessBuilder</code>	<code>StringBuffer</code>
<code>Character</code>	<code>InheritableThreadLocal</code>	<code>ProcessBuilder.Redirect</code>	<code>StringBuilder</code>
<code>Character.Subset</code>	<code>Integer</code>	<code>Runtime</code>	<code>System</code>
<code>Character.UnicodeBlock</code>	<code>Long</code>	<code>RuntimePermission</code>	<code>Thread</code>
<code>Class</code>	<code>Math</code>	<code>SecurityManager</code>	<code>ThreadGroup</code>
<code>ClassLoader</code>	<code>Number</code>	<code>Short</code>	<code>ThreadLocal</code>
<code>ClassValue</code>	<code>Object</code>	<code>StackTraceElement</code>	<code>Throwable</code>
<code>Compiler</code>	<code>Package</code>	<code>StrictMath</code>	<code>Void</code>
<code>Double</code>			

`java.lang` defines the following interfaces:

<code>Appendable</code>	<code>Cloneable</code>	<code>Readable</code>
<code>AutoCloseable</code>	<code>Comparable</code>	<code>Runnable</code>
<code>CharSequence</code>	<code>Iterable</code>	<code>Thread.UncaughtExceptionHandler</code>

Several of the classes contained in `java.lang` contain deprecated methods, most dating back to Java 1.0. These deprecated methods are still provided by Java to support an ever-shrinking pool of legacy code and are not recommended for new code. Because of this, the deprecated methods are not discussed here.

Primitive Type Wrappers

As mentioned in Part I of this book, Java uses primitive types, such as **int** and **char**, for performance reasons. These data types are not part of the object hierarchy. They are passed by value to methods and cannot be directly passed by reference. Also, there is no way for two methods to refer to the *same instance* of an **int**. At times, you will need to create an object representation for one of these primitive types. For example, there are collection classes discussed in Chapter 18 that deal only with objects; to store a primitive type in one of these classes, you need to wrap the primitive type in a class. To address this need, Java provides classes that correspond to each of the primitive types. In essence, these classes encapsulate, or *wrap*, the primitive types within a class. Thus, they are commonly referred to as *type wrappers*. The type wrappers were introduced in Chapter 12. They are examined in detail here.

Number

The abstract class **Number** defines a superclass that is implemented by the classes that wrap the numeric types **byte**, **short**, **int**, **long**, **float**, and **double**. **Number** has abstract methods that return the value of the object in each of the different number formats. For example, **doubleValue()** returns the value as a **double**, **floatValue()** returns the value as a **float**, and so on. These methods are shown here:

```
byte byteValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
short shortValue()
```

The values returned by these methods might be rounded, truncated, or result in a “garbage” value due to the effects of a narrowing conversion.

Number has concrete subclasses that hold explicit values of each primitive numeric type: **Double**, **Float**, **Byte**, **Short**, **Integer**, and **Long**.

Double and Float

Double and **Float** are wrappers for floating-point values of type **double** and **float**, respectively. The constructors for **Float** are shown here:

```
Float(double num)
Float(float num)
Float(String str) throws NumberFormatException
```

As you can see, **Float** objects can be constructed with values of type **float** or **double**. They can also be constructed from the string representation of a floating-point number.

The constructors for **Double** are shown here:

```
Double(double num)
Double(String str) throws NumberFormatException
```

Double objects can be constructed with a **double** value or a string containing a floating-point value.

The methods defined by **Float** include those shown in Table 17-1. The methods defined by **Double** include those shown in Table 17-2. Both **Float** and **Double** define the following constants:

BYTES	The width of a float or double in bytes (Added by JDK 8.)
MAX_EXPONENT	Maximum exponent
MAX_VALUE	Maximum positive value
MIN_EXPONENT	Minimum exponent
MIN_NORMAL	Minimum positive normal value
MIN_VALUE	Minimum positive value
NaN	Not a number
POSITIVE_INFINITY	Positive infinity
NEGATIVE_INFINITY	Negative infinity
SIZE	The bit width of the wrapped value
TYPE	The Class object for float or double

Method	Description
byte byteValue()	Returns the value of the invoking object as a byte .
static int compare(float num1, float num2)	Compares the values of <i>num1</i> and <i>num2</i> . Returns 0 if the values are equal. Returns a negative value if <i>num1</i> is less than <i>num2</i> . Returns a positive value if <i>num1</i> is greater than <i>num2</i> .
int compareTo(Float <i>f</i>)	Compares the numerical value of the invoking object with that of <i>f</i> . Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
double doubleValue()	Returns the value of the invoking object as a double .
boolean equals(Object <i>FloatObj</i>)	Returns true if the invoking Float object is equivalent to <i>FloatObj</i> . Otherwise, it returns false .
static int floatToIntBits(float <i>num</i>)	Returns the IEEE-compatible, single-precision bit pattern that corresponds to <i>num</i> .
static int floatToRawIntBits(float <i>num</i>)	Returns the IEEE-compatible single-precision bit pattern that corresponds to <i>num</i> . A NaN value is preserved.
float floatValue()	Returns the value of the invoking object as a float .
int hashCode()	Returns the hash code for the invoking object.
static int hashCode(float <i>num</i>)	Returns the hash code for <i>num</i> . (Added by JDK 8.)
static float intBitsToFloat(int <i>num</i>)	Returns float equivalent of the IEEE-compatible, single-precision bit pattern specified by <i>num</i> .

Table 17-1 The Methods Defined by **Float**

Method	Description
int intValue()	Returns the value of the invoking object as an int .
static boolean isFinite(float num)	Returns true if <i>num</i> is not NaN and is not infinite. (Added by JDK 8.)
boolean isInfinite()	Returns true if the invoking object contains an infinite value. Otherwise, it returns false .
static boolean isInfinite(float num)	Returns true if <i>num</i> specifies an infinite value. Otherwise, it returns false .
boolean isNaN()	Returns true if the invoking object contains a value that is not a number. Otherwise, it returns false .
static boolean isNaN(float num)	Returns true if <i>num</i> specifies a value that is not a number. Otherwise, it returns false .
long longValue()	Returns the value of the invoking object as a long .
static float max(float val, float val2)	Returns the maximum of <i>val</i> and <i>val2</i> . (Added by JDK 8.)
static float min(float val, float val2)	Returns the minimum of <i>val</i> and <i>val2</i> . (Added by JDK 8.)
static float parseFloat(String str) throws NumberFormatException	Returns the float equivalent of the number contained in the string specified by <i>str</i> using radix 10.
short shortValue()	Returns the value of the invoking object as a short .
static float sum(float val, float val2)	Returns the result of <i>val</i> + <i>val2</i> . (Added by JDK 8.)
static String toHexString(float num)	Returns a string containing the value of <i>num</i> in hexadecimal format.
String toString()	Returns the string equivalent of the invoking object.
static String toString(float num)	Returns the string equivalent of the value specified by <i>num</i> .
static Float valueOf(float num)	Returns a Float object containing the value passed in <i>num</i> .
static Float valueOf(String str) throws NumberFormatException	Returns the Float object that contains the value specified by the string in <i>str</i> .

Table 17-1 The Methods Defined by **Float** (*continued*)

Method	Description
byte byteValue()	Returns the value of the invoking object as a byte .
static int compare(double num1, double num2)	Compares the values of <i>num1</i> and <i>num2</i> . Returns 0 if the values are equal. Returns a negative value if <i>num1</i> is less than <i>num2</i> . Returns a positive value if <i>num1</i> is greater than <i>num2</i> .

Table 17-2 The Methods Defined by **Double**

Method	Description
int compareTo(Double <i>d</i>)	C.compares the numerical value of the invoking object with that of <i>d</i> . Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static long doubleToLongBits(double <i>num</i>)	Returns the IEEE-compatible, double-precision bit pattern that corresponds to <i>num</i> .
static long doubleToRawLongBits(double <i>num</i>)	Returns the IEEE-compatible double-precision bit pattern that corresponds to <i>num</i> . A NaN value is preserved.
double doubleValue()	Returns the value of the invoking object as a double .
boolean equals(Object <i>DoubleObj</i>)	Returns true if the invoking Double object is equivalent to <i>DoubleObj</i> . Otherwise, it returns false .
float floatValue()	Returns the value of the invoking object as a float .
int hashCode()	Returns the hash code for the invoking object.
static int hashCode(double <i>num</i>)	Returns the hash code for <i>num</i> . (Added by JDK 8.)
int intValue()	Returns the value of the invoking object as an int .
static boolean isFinite(double <i>num</i>)	Returns true if <i>num</i> is not NaN and is not infinite. (Added by JDK 8.)
boolean isInfinite()	Returns true if the invoking object contains an infinite value. Otherwise, it returns false .
static boolean isInfinite(double <i>num</i>)	Returns true if <i>num</i> specifies an infinite value. Otherwise, it returns false .
boolean isNaN()	Returns true if the invoking object contains a value that is not a number. Otherwise, it returns false .
static boolean isNaN(double <i>num</i>)	Returns true if <i>num</i> specifies a value that is not a number. Otherwise, it returns false .
static double longBitsToDouble(long <i>num</i>)	Returns double equivalent of the IEEE-compatible, double-precision bit pattern specified by <i>num</i> .
long longValue()	Returns the value of the invoking object as a long .
static double max(double <i>val</i> , double <i>val2</i>)	Returns the maximum of <i>val</i> and <i>val2</i> . (Added by JDK 8.)
static double min(double <i>val</i> , double <i>val2</i>)	Returns the minimum of <i>val</i> and <i>val2</i> . (Added by JDK 8.)
static double parseDouble(String <i>str</i>) throws NumberFormatException	Returns the double equivalent of the number contained in the string specified by <i>str</i> using radix 10.

Table 17-2 The Methods Defined by **Double** (*continued*)

Method	Description
short shortValue()	Returns the value of the invoking object as a short .
static double sum(double <i>val</i> , double <i>val2</i>)	Returns the result of <i>val</i> + <i>val2</i> . (Added by JDK 8.)
static String toHexString(double <i>num</i>)	Returns a string containing the value of <i>num</i> in hexadecimal format.
String toString()	Returns the string equivalent of the invoking object.
static String toString(double <i>num</i>)	Returns the string equivalent of the value specified by <i>num</i> .
static Double valueOf(double <i>num</i>)	Returns a Double object containing the value passed in <i>num</i> .
static Double valueOf(String <i>str</i>) throws NumberFormatException	Returns a Double object that contains the value specified by the string in <i>str</i> .

Table 17-2 The Methods Defined by **Double** (*continued*)

The following example creates two **Double** objects—one by using a **double** value and the other by passing a string that can be parsed as a **double**:

```
class DoubleDemo {
    public static void main(String args[]) {
        Double d1 = new Double(3.14159);
        Double d2 = new Double("314159E-5");

        System.out.println(d1 + " = " + d2 + " -> " + d1.equals(d2));
    }
}
```

As you can see from the following output, both constructors created identical **Double** instances, as shown by the **equals()** method returning **true**:

```
3.14159 = 3.14159 -> true
```

Understanding **isInfinite()** and **isNaN()**

Float and **Double** provide the methods **isInfinite()** and **isNaN()**, which help when manipulating two special **double** and **float** values. These methods test for two unique values defined by the IEEE floating-point specification: infinity and NaN (not a number). **isInfinite()** returns **true** if the value being tested is infinitely large or small in magnitude. **isNaN()** returns **true** if the value being tested is not a number.

The following example creates two **Double** objects; one is infinite, and the other is not a number:

```
// Demonstrate isInfinite() and isNaN()
class InfNaN {
```

```

public static void main(String args[]) {
    Double d1 = new Double(1/0.);
    Double d2 = new Double(0/0.);

    System.out.println(d1 + ": " + d1.isInfinite() + ", " + d1isNaN());
    System.out.println(d2 + ": " + d2.isInfinite() + ", " + d2.isnan());
}
}

```

This program generates the following output:

```

Infinity: true, false
NaN: false, true

```

Byte, Short, Integer, and Long

The **Byte**, **Short**, **Integer**, and **Long** classes are wrappers for **byte**, **short**, **int**, and **long** integer types, respectively. Their constructors are shown here:

Byte(byte num)

Byte(String str) throws NumberFormatException

Short(short num)

Short(String str) throws NumberFormatException

Integer(int num)

Integer(String str) throws NumberFormatException

Long(long num)

Long(String str) throws NumberFormatException

As you can see, these objects can be constructed from numeric values or from strings that contain valid whole number values.

The methods defined by these classes are shown in Tables 17-3 through 17-6. As you can see, they define methods for parsing integers from strings and converting strings back into integers. Variants of these methods allow you to specify the *radix*, or numeric base, for conversion. Common radices are 2 for binary, 8 for octal, 10 for decimal, and 16 for hexadecimal.

The following constants are defined:

BYTES	The width of the integer type in bytes (Added by JDK 8.)
MIN_VALUE	Minimum value
MAX_VALUE	Maximum value
SIZE	The bit width of the wrapped value
TYPE	The Class object for byte , short , int , or long

Method	Description
byte byteValue()	Returns the value of the invoking object as a byte .
static int compare(byte <i>num1</i> , byte <i>num2</i>)	Compares the values of <i>num1</i> and <i>num2</i> . Returns 0 if the values are equal. Returns a negative value if <i>num1</i> is less than <i>num2</i> . Returns a positive value if <i>num1</i> is greater than <i>num2</i> .
int compareTo(Byte <i>b</i>)	Compares the numerical value of the invoking object with that of <i>b</i> . Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static Byte decode(String <i>str</i>) throws NumberFormatException	Returns a Byte object that contains the value specified by the string in <i>str</i> .
double doubleValue()	Returns the value of the invoking object as a double .
boolean equals(Object <i>ByteObj</i>)	Returns true if the invoking Byte object is equivalent to <i>ByteObj</i> . Otherwise, it returns false .
float floatValue()	Returns the value of the invoking object as a float .
int hashCode()	Returns the hash code for the invoking object.
static int hashCode(byte <i>num</i>)	Returns the hash code for <i>num</i> . (Added by JDK 8.)
int intValue()	Returns the value of the invoking object as an int .
long longValue()	Returns the value of the invoking object as a long .
static byte parseByte(String <i>str</i>) throws NumberFormatException	Returns the byte equivalent of the number contained in the string specified by <i>str</i> using radix 10.
static byte parseByte(String <i>str</i> , int <i>radix</i>) throws NumberFormatException	Returns the byte equivalent of the number contained in the string specified by <i>str</i> using the specified radix.
short shortValue()	Returns the value of the invoking object as a short .
String toString()	Returns a string that contains the decimal equivalent of the invoking object.
static String toString(byte <i>num</i>)	Returns a string that contains the decimal equivalent of <i>num</i> .
static int toUnsignedInt(byte <i>val</i>)	Returns the value of <i>val</i> as an unsigned integer. (Added by JDK 8.)
static long toUnsignedLong(byte <i>val</i>)	Returns the value of <i>val</i> as an unsigned long integer. (Added by JDK 8.)
static Byte valueOf(byte <i>num</i>)	Returns a Byte object containing the value passed in <i>num</i> .
static Byte valueOf(String <i>str</i>) throws NumberFormatException	Returns a Byte object that contains the value specified by the string in <i>str</i> .
static Byte valueOf(String <i>str</i> , int <i>radix</i>) throws NumberFormatException	Returns a Byte object that contains the value specified by the string in <i>str</i> using the specified <i>radix</i> .

Table 17-3 The Methods Defined by **Byte**

Method	Description
byte <code>byteValue()</code>	Returns the value of the invoking object as a byte .
static int <code>compare(short num1, short num2)</code>	Compares the values of <i>num1</i> and <i>num2</i> . Returns 0 if the values are equal. Returns a negative value if <i>num1</i> is less than <i>num2</i> . Returns a positive value if <i>num1</i> is greater than <i>num2</i> .
int <code>compareTo(Short s)</code>	Compares the numerical value of the invoking object with that of <i>s</i> . Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static Short <code>decode(String str) throws NumberFormatException</code>	Returns a Short object that contains the value specified by the string in <i>str</i> .
double <code>doubleValue()</code>	Returns the value of the invoking object as a double .
boolean <code>equals(Object ShortObj)</code>	Returns true if the invoking Short object is equivalent to <i>ShortObj</i> . Otherwise, it returns false .
float <code>floatValue()</code>	Returns the value of the invoking object as a float .
int <code>hashCode()</code>	Returns the hash code for the invoking object.
static int <code>hashCode(short num)</code>	Returns the hash code for <i>num</i> . (Added by JDK 8.)
int <code>intValue()</code>	Returns the value of the invoking object as an int .
long <code>longValue()</code>	Returns the value of the invoking object as a long .
static short <code>parseShort(String str) throws NumberFormatException</code>	Returns the short equivalent of the number contained in the string specified by <i>str</i> using radix 10.
static short <code>parseShort(String str, int radix) throws NumberFormatException</code>	Returns the short equivalent of the number contained in the string specified by <i>str</i> using the specified <i>radix</i> .
static short <code>reverseBytes(short num)</code>	Exchanges the high- and low-order bytes of <i>num</i> and returns the result.
short <code>shortValue()</code>	Returns the value of the invoking object as a short .
String <code>toString()</code>	Returns a string that contains the decimal equivalent of the invoking object.
static String <code>toString(short num)</code>	Returns a string that contains the decimal equivalent of <i>num</i> .
static int <code>toUnsignedInt(short val)</code>	Returns the value of <i>val</i> as an unsigned integer. (Added by JDK 8.)
static long <code>toUnsignedLong(short val)</code>	Returns the value of <i>val</i> as an unsigned long integer. (Added by JDK 8.)
static Short <code>valueOf(short num)</code>	Returns a Short object containing the value passed in <i>num</i> .
static Short <code>valueOf(String str) throws NumberFormatException</code>	Returns a Short object that contains the value specified by the string in <i>str</i> using radix 10.
static Short <code>valueOf(String str, int radix) throws NumberFormatException</code>	Returns a Short object that contains the value specified by the string in <i>str</i> using the specified <i>radix</i> .

Table 17-4 The Methods Defined by **Short**

Method	Description
static int bitCount(int <i>num</i>)	Returns the number of set bits in <i>num</i> .
byte byteValue()	Returns the value of the invoking object as a byte .
static int compare(int <i>num1</i> , int <i>num2</i>)	Compares the values of <i>num1</i> and <i>num2</i> . Returns 0 if the values are equal. Returns a negative value if <i>num1</i> is less than <i>num2</i> . Returns a positive value if <i>num1</i> is greater than <i>num2</i> .
int compareTo(Integer <i>i</i>)	Compares the numerical value of the invoking object with that of <i>i</i> . Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compareUnsigned(int <i>num1</i> , int <i>num2</i>)	Performs an unsigned comparison of <i>num1</i> and <i>num2</i> . Returns 0 if the values are equal. Returns a negative value if <i>num1</i> is less than <i>num2</i> . Returns a positive value if <i>num1</i> is greater than <i>num2</i> . (Added by JDK 8.)
static Integer decode(String <i>str</i>) throws NumberFormatException	Returns an Integer object that contains the value specified by the string in <i>str</i> .
static int divideUnsigned(int <i>dividend</i> , int <i>divisor</i>)	Returns the result, as an unsigned value, of the unsigned division of <i>dividend</i> by <i>divisor</i> . (Added by JDK 8.)
double doubleValue()	Returns the value of the invoking object as a double .
boolean equals(Object <i>IntegerObj</i>)	Returns true if the invoking Integer object is equivalent to <i>IntegerObj</i> . Otherwise, it returns false .
float floatValue()	Returns the value of the invoking object as a float .
static Integer getInteger(String <i>propertyName</i>)	Returns the value associated with the environmental property specified by <i>propertyName</i> . A null is returned on failure.
static Integer getInteger(String <i>propertyName</i> , int <i>default</i>)	Returns the value associated with the environmental property specified by <i>propertyName</i> . The value of <i>default</i> is returned on failure.
static Integer getInteger(String <i>propertyName</i> , Integer <i>default</i>)	Returns the value associated with the environmental property specified by <i>propertyName</i> . The value of <i>default</i> is returned on failure.
int hashCode()	Returns the hash code for the invoking object.
static int hashCode(int <i>num</i>)	Returns the hash code for <i>num</i> . (Added by JDK 8.)
static int highestOneBit(int <i>num</i>)	Determines the position of the highest order set bit in <i>num</i> . It returns a value in which only this bit is set. If no bit is set to one, then zero is returned.
int intValue()	Returns the value of the invoking object as an int .
long longValue()	Returns the value of the invoking object as a long .
static int lowestOneBit(int <i>num</i>)	Determines the position of the lowest order set bit in <i>num</i> . It returns a value in which only this bit is set. If no bit is set to one, then zero is returned.
static int max(int <i>val</i> , int <i>val2</i>)	Returns the maximum of <i>val</i> and <i>val2</i> . (Added by JDK 8.)
static int min(int <i>val</i> , int <i>val2</i>)	Returns the minimum of <i>val</i> and <i>val2</i> . (Added by JDK 8.)
static int numberOfLeadingZeros(int <i>num</i>)	Returns the number of high-order zero bits that precede the first high-order set bit in <i>num</i> . If <i>num</i> is zero, 32 is returned.

Table 17-5 The Methods Defined by **Integer**

Method	Description
static int numberOfTrailingZeros(int num)	Returns the number of low-order zero bits that precede the first low-order set bit in <i>num</i> . If <i>num</i> is zero, 32 is returned.
static int parseInt(String str) throws NumberFormatException	Returns the integer equivalent of the number contained in the string specified by <i>str</i> using radix 10.
static int parseInt(String str, int radix) throws NumberFormatException	Returns the integer equivalent of the number contained in the string specified by <i>str</i> using the specified <i>radix</i> .
static int parseUnsignedInt(String str) throws NumberFormatException	Returns the unsigned integer equivalent of the number contained in the string specified by <i>str</i> using the radix 10. (Added by JDK 8.)
static int parseUnsignedInt(String str, int radix) throws NumberFormatException	Returns the unsigned integer equivalent of the number contained in the string specified by <i>str</i> using the radix specified by <i>radix</i> . (Added by JDK 8.)
static int remainderUnsigned(int dividend, int divisor)	Returns the remainder, as an unsigned value, of the unsigned division of <i>dividend</i> by <i>divisor</i> . (Added by JDK 8.)
static int reverse(int num)	Reverses the order of the bits in <i>num</i> and returns the result.
static int reverseBytes(int num)	Reverses the order of the bytes in <i>num</i> and returns the result.
static int rotateLeft(int num, int n)	Returns the result of rotating <i>num</i> left <i>n</i> positions.
static int rotateRight(int num, int n)	Returns the result of rotating <i>num</i> right <i>n</i> positions.
short shortValue()	Returns the value of the invoking object as a short .
static int signum(int num)	Returns -1 if <i>num</i> is negative, 0 if it is zero, and 1 if it is positive.
static int sum(int val, int val2)	Returns the result of <i>val</i> + <i>val2</i> . (Added by JDK 8.)
static String toBinaryString(int num)	Returns a string that contains the binary equivalent of <i>num</i> .
static String toHexString(int num)	Returns a string that contains the hexadecimal equivalent of <i>num</i> .
static String toOctalString(int num)	Returns a string that contains the octal equivalent of <i>num</i> .
String toString()	Returns a string that contains the decimal equivalent of the invoking object.
static String toString(int num)	Returns a string that contains the decimal equivalent of <i>num</i> .
static String toString(int num, int radix)	Returns a string that contains the decimal equivalent of <i>num</i> using the specified <i>radix</i> .
static long toUnsignedLong(int val)	Returns the value of <i>val</i> as an unsigned long integer. (Added by JDK 8.)
static String toUnsignedString(int val)	Returns a string that contains the decimal value of <i>val</i> as an unsigned integer. (Added by JDK 8.)
static String toUnsignedString(int val, int radix)	Returns a string that contains the value of <i>val</i> as an unsigned integer in the radix specified by <i>radix</i> . (Added by JDK 8.)
static Integer valueOf(int num)	Returns an Integer object containing the value passed in <i>num</i> .
static Integer valueOf(String str) throws NumberFormatException	Returns an Integer object that contains the value specified by the string in <i>str</i> .
static Integer valueOf(String str, int radix) throws NumberFormatException	Returns an Integer object that contains the value specified by the string in <i>str</i> using the specified <i>radix</i> .

Table 17-5 The Methods Defined by **Integer** (continued)

Method	Description
static int bitCount(long <i>num</i>)	Returns the number of set bits in <i>num</i> .
byte byteValue()	Returns the value of the invoking object as a byte .
static int compare(long <i>num1</i> , long <i>num2</i>)	Compares the values of <i>num1</i> and <i>num2</i> . Returns 0 if the values are equal. Returns a negative value if <i>num1</i> is less than <i>num2</i> . Returns a positive value if <i>num1</i> is greater than <i>num2</i> .
int compareTo(Long <i>l</i>)	Compares the numerical value of the invoking object with that of <i>l</i> . Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compareUnsigned(long <i>num1</i> , long <i>num2</i>)	Performs an unsigned comparison of <i>num1</i> and <i>num2</i> . Returns 0 if the values are equal. Returns a negative value if <i>num1</i> is less than <i>num2</i> . Returns a positive value if <i>num1</i> is greater than <i>num2</i> . (Added by JDK 8.)
static Long decode(String <i>str</i>) throws NumberFormatException	Returns a Long object that contains the value specified by the string in <i>str</i> .
static long divideUnsigned(long <i>dividend</i> , long <i>divisor</i>)	Returns the result, as an unsigned value, of the unsigned division of <i>dividend</i> by <i>divisor</i> . (Added by JDK 8.)
double doubleValue()	Returns the value of the invoking object as a double .
boolean equals(Object <i>LongObj</i>)	Returns true if the invoking Long object is equivalent to <i>LongObj</i> . Otherwise, it returns false .
float floatValue()	Returns the value of the invoking object as a float .
static Long getLong(String <i>propertyName</i>)	Returns the value associated with the environmental property specified by <i>propertyName</i> . A null is returned on failure.
static Long getLong(String <i>propertyName</i> , long <i>default</i>)	Returns the value associated with the environmental property specified by <i>propertyName</i> . The value of <i>default</i> is returned on failure.
static Long getLong(String <i>propertyName</i> , Long <i>default</i>)	Returns the value associated with the environmental property specified by <i>propertyName</i> . The value of <i>default</i> is returned on failure.
int hashCode()	Returns the hash code for the invoking object.
static int hashCode(long <i>num</i>)	Returns the hash code for <i>num</i> . (Added by JDK 8.)
static long highestOneBit(long <i>num</i>)	Determines the position of the highest-order set bit in <i>num</i> . It returns a value in which only this bit is set. If no bit is set to one, then zero is returned.
int intValue()	Returns the value of the invoking object as an int .
long longValue()	Returns the value of the invoking object as a long .
static long lowestOneBit(long <i>num</i>)	Determines the position of the lowest-order set bit in <i>num</i> . It returns a value in which only this bit is set. If no bit is set to one, then zero is returned.
static long max(long <i>val</i> , long <i>val2</i>)	Returns the maximum of <i>val</i> and <i>val2</i> . (Added by JDK 8.)
static long min(long <i>val</i> , long <i>val2</i>)	Returns the minimum of <i>val</i> and <i>val2</i> . (Added by JDK 8.)
static int numberOfLeadingZeros(long <i>num</i>)	Returns the number of high-order zero bits that precede the first high-order set bit in <i>num</i> . If <i>num</i> is zero, 64 is returned.

Table 17-6 The Methods Defined by **Long**

Method	Description
static int numberOfTrailingZeros(long <i>num</i>)	Returns the number of low-order zero bits that precede the first low-order set bit in <i>num</i> . If <i>num</i> is zero, 64 is returned.
static long parseLong(String <i>str</i>) throws NumberFormatException	Returns the long equivalent of the number contained in the string specified by <i>str</i> using radix 10.
static long parseLong(String <i>str</i> , int <i>radix</i>) throws NumberFormatException	Returns the long equivalent of the number contained in the string specified by <i>str</i> using the specified <i>radix</i> .
static long parseUnsignedLong(String <i>str</i>) throws NumberFormatException	Returns the unsigned integer equivalent of the number contained in the string specified by <i>str</i> using the radix 10. (Added by JDK 8.)
static long parseUnsignedLong(String <i>str</i> , int <i>radix</i>) throws NumberFormatException	Returns the unsigned integer equivalent of the number contained in the string specified by <i>str</i> using the radix specified by <i>radix</i> . (Added by JDK 8.)
static long remainderUnsigned(long <i>dividend</i> , long <i>divisor</i>)	Returns the remainder, as an unsigned value, of the unsigned division of <i>dividend</i> by <i>divisor</i> . (Added by JDK 8.)
static long reverse(long <i>num</i>)	Reverses the order of the bits in <i>num</i> and returns the result.
static long reverseBytes(long <i>num</i>)	Reverses the order of the bytes in <i>num</i> and returns the result.
static long rotateLeft(long <i>num</i> , int <i>n</i>)	Returns the result of rotating <i>num</i> left <i>n</i> positions.
static long rotateRight(long <i>num</i> , int <i>n</i>)	Returns the result of rotating <i>num</i> right <i>n</i> positions.
short shortValue()	Returns the value of the invoking object as a short .
static int signum(long <i>num</i>)	Returns -1 if <i>num</i> is negative, 0 if it is zero, and 1 if it is positive.
static long sum(long <i>val</i> , long <i>val2</i>)	Returns the result of <i>val</i> + <i>val2</i> . (Added by JDK 8.)
static String toBinaryString(long <i>num</i>)	Returns a string that contains the binary equivalent of <i>num</i> .
static String toHexString(long <i>num</i>)	Returns a string that contains the hexadecimal equivalent of <i>num</i> .
static String toOctalString(long <i>num</i>)	Returns a string that contains the octal equivalent of <i>num</i> .
String toString()	Returns a string that contains the decimal equivalent of the invoking object.
static String toString(long <i>num</i>)	Returns a string that contains the decimal equivalent of <i>num</i> .
static String toString(long <i>num</i> , int <i>radix</i>)	Returns a string that contains the decimal equivalent of <i>num</i> using the specified <i>radix</i> .
static String toUnsignedString(long <i>val</i>)	Returns a string that contains the decimal value of <i>val</i> as an unsigned integer. (Added by JDK 8.)
static String toUnsignedString(long <i>val</i> , int <i>radix</i>)	Returns a string that contains the value of <i>val</i> as an unsigned integer in the radix specified by <i>radix</i> . (Added by JDK 8.)
static Long valueOf(long <i>num</i>)	Returns a Long object containing the value passed in <i>num</i> .
static Long valueOf(String <i>str</i>) throws NumberFormatException	Returns a Long object that contains the value specified by the string in <i>str</i> .
static Long valueOf(String <i>str</i> , int <i>radix</i>) throws NumberFormatException	Returns a Long object that contains the value specified by the string in <i>str</i> using the specified <i>radix</i> .

Table 17-6 The Methods Defined by **Long** (*continued*)

Converting Numbers to and from Strings

One of the most common programming chores is converting the string representation of a number into its internal, binary format. Fortunately, Java provides an easy way to accomplish this. The **Byte**, **Short**, **Integer**, and **Long** classes provide the **parseByte()**, **parseShort()**, **parseInt()**, and **parseLong()** methods, respectively. These methods return the **byte**, **short**, **int**, or **long** equivalent of the numeric string with which they are called. (Similar methods also exist for the **Float** and **Double** classes.)

The following program demonstrates **parseInt()**. It sums a list of integers entered by the user. It reads the integers using **readLine()** and uses **parseInt()** to convert these strings into their **int** equivalents.

```
/* This program sums a list of numbers entered
   by the user.  It converts the string representation
   of each number into an int using parseInt().
*/
import java.io.*;

class ParseDemo {
    public static void main(String args[])
        throws IOException
    {
        // create a BufferedReader using System.in
        BufferedReader br = new
            BufferedReader(new InputStreamReader(System.in));
        String str;
        int i;
        int sum=0;

        System.out.println("Enter numbers, 0 to quit.");
        do {
            str = br.readLine();
            try {
                i = Integer.parseInt(str);
            } catch(NumberFormatException e) {
                System.out.println("Invalid format");
                i = 0;
            }
            sum += i;
            System.out.println("Current sum is: " + sum);
        } while(i != 0);
    }
}
```

To convert a whole number into a decimal string, use the versions of **toString()** defined in the **Byte**, **Short**, **Integer**, or **Long** classes. The **Integer** and **Long** classes also provide the

methods **toBinaryString()**, **toHexString()**, and **toOctalString()**, which convert a value into a binary, hexadecimal, or octal string, respectively.

The following program demonstrates binary, hexadecimal, and octal conversion:

```
/* Convert an integer into binary, hexadecimal,
   and octal.
*/
class StringConversions {
    public static void main(String args[]) {
        int num = 19648;
        System.out.println(num + " in binary: " +
                           Integer.toBinaryString(num));

        System.out.println(num + " in octal: " +
                           Integer.toOctalString(num));

        System.out.println(num + " in hexadecimal: " +
                           Integer.toHexString(num));
    }
}
```

The output of this program is shown here:

```
19648 in binary: 100110011000000
19648 in octal: 46300
19648 in hexadecimal: 4cc0
```

Character

Character is a simple wrapper around a **char**. The constructor for **Character** is

`Character(char ch)`

Here, *ch* specifies the character that will be wrapped by the **Character** object being created.

To obtain the **char** value contained in a **Character** object, call **charValue()**, shown here:

`char charValue()`

It returns the character.

The **Character** class defines several constants, including the following:

BYTES	The width of a char in bytes (Added by JDK 8.)
MAX_RADIX	The largest radix
MIN_RADIX	The smallest radix
MAX_VALUE	The largest character value
MIN_VALUE	The smallest character value
TYPE	The Class object for char

Character includes several static methods that categorize characters and alter their case. A sampling is shown in Table 17-7. The following example demonstrates several of these methods:

```
// Demonstrate several Is... methods.

class IsDemo {
    public static void main(String args[]) {
        char a[] = {'a', 'b', '5', '?', 'A', ' '};

        for(int i=0; i<a.length; i++) {
            if(Character.isDigit(a[i]))
                System.out.println(a[i] + " is a digit.");
            if(Character.isLetter(a[i]))
                System.out.println(a[i] + " is a letter.");
            if(Character.isWhitespace(a[i]))
                System.out.println(a[i] + " is whitespace.");
            if(Character.isUpperCase(a[i]))
                System.out.println(a[i] + " is uppercase.");
            if(Character.isLowerCase(a[i]))
                System.out.println(a[i] + " is lowercase.");
        }
    }
}
```

The output from this program is shown here:

```
a is a letter.
a is lowercase.
b is a letter.
b is lowercase.
5 is a digit.
A is a letter.
A is uppercase.
is whitespace.
```

Character defines two methods, **forDigit()** and **digit()**, that enable you to convert between integer values and the digits they represent. They are shown here:

```
static char forDigit(int num, int radix)
static int digit(char digit, int radix)
```

forDigit() returns the digit character associated with the value of *num*. The radix of the conversion is specified by *radix*. **digit()** returns the integer value associated with the specified character (which is presumably a digit) according to the specified radix. (There is a second form of **digit()** that takes a code point. See the following section for a discussion of code points.)

Another method defined by **Character** is **compareTo()**, which has the following form:

```
int compareTo(Character c)
```

It returns zero if the invoking object and *c* have the same value. It returns a negative value if the invoking object has a lower value. Otherwise, it returns a positive value.

Method	Description
static boolean isDefined(char <i>ch</i>)	Returns true if <i>ch</i> is defined by Unicode. Otherwise, it returns false .
static boolean isDigit(char <i>ch</i>)	Returns true if <i>ch</i> is a digit. Otherwise, it returns false .
static boolean isIdentifierIgnorable(char <i>ch</i>)	Returns true if <i>ch</i> should be ignored in an identifier. Otherwise, it returns false .
static boolean isISOControl(char <i>ch</i>)	Returns true if <i>ch</i> is an ISO control character. Otherwise, it returns false .
static boolean isJavaIdentifierPart(char <i>ch</i>)	Returns true if <i>ch</i> is allowed as part of a Java identifier (other than the first character). Otherwise, it returns false .
static boolean isJavaIdentifierStart(char <i>ch</i>)	Returns true if <i>ch</i> is allowed as the first character of a Java identifier. Otherwise, it returns false .
static boolean isLetter(char <i>ch</i>)	Returns true if <i>ch</i> is a letter. Otherwise, it returns false.
static boolean isLetterOrDigit(char <i>ch</i>)	Returns true if <i>ch</i> is a letter or a digit. Otherwise, it returns false .
static boolean isLowerCase(char <i>ch</i>)	Returns true if <i>ch</i> is a lowercase letter. Otherwise, it returns false .
static boolean isMirrored(char <i>ch</i>)	Returns true if <i>ch</i> is a mirrored Unicode character. A mirrored character is one that is reversed for text that is displayed right-to-left.
static boolean isSpaceChar(char <i>ch</i>)	Returns true if <i>ch</i> is a Unicode space character. Otherwise, it returns false .
static boolean isTitleCase(char <i>ch</i>)	Returns true if <i>ch</i> is a Unicode titlecase character. Otherwise, it returns false .
static boolean isUnicodeIdentifierPart(char <i>ch</i>)	Returns true if <i>ch</i> is allowed as part of a Unicode identifier (other than the first character). Otherwise, it returns false .
static Boolean isUnicodeIdentifierStart(char <i>ch</i>)	Returns true if <i>ch</i> is allowed as the first character of a Unicode identifier. Otherwise, it returns false .
static boolean isUpperCase(char <i>ch</i>)	Returns true if <i>ch</i> is an uppercase letter. Otherwise, it returns false .
static boolean isWhitespace(char <i>ch</i>)	Returns true if <i>ch</i> is whitespace. Otherwise, it returns false .
static char toLowerCase(char <i>ch</i>)	Returns lowercase equivalent of <i>ch</i> .
static char toTitleCase(char <i>ch</i>)	Returns titlecase equivalent of <i>ch</i> .
static char toUpperCase(char <i>ch</i>)	Returns uppercase equivalent of <i>ch</i> .

Table 17-7 Various **Character** Methods

Character includes a method called `getDirectionality()` which can be used to determine the direction of a character. Several constants are defined that describe directionality. Most programs will not need to use character directionality.

Character also overrides the `equals()` and `hashCode()` methods.

Two other character-related classes are **Character.Subset**, used to describe a subset of Unicode, and **Character.UnicodeBlock**, which contains Unicode character blocks.

Additions to Character for Unicode Code Point Support

Relatively recently, major additions were made to **Character**. Beginning with JDK 5, the **Character** class has included support for 32-bit Unicode characters. In the past, all Unicode characters could be held by 16 bits, which is the size of a **char** (and the size of the value encapsulated within a **Character**), because those values ranged from 0 to FFFF. However, the Unicode character set has been expanded, and more than 16 bits are required. Characters can now range from 0 to 10FFFF.

Here are three important terms. A *code point* is a character in the range 0 to 10FFFF. Characters that have values greater than FFFF are called *supplemental characters*. The *basic multilingual plane (BMP)* are those characters between 0 and FFFF.

The expansion of the Unicode character set caused a fundamental problem for Java. Because a supplemental character has a value greater than a **char** can hold, some means of handling the supplemental characters was needed. Java addressed this problem in two ways. First, Java uses two **chars** to represent a supplemental character. The first **char** is called the *high surrogate*, and the second is called the *low surrogate*. New methods, such as `codePointAt()`, were provided to translate between code points and supplemental characters.

Secondly, Java overloaded several preexisting methods in the **Character** class. The overloaded forms use **int** rather than **char** data. Because an **int** is large enough to hold any character as a single value, it can be used to store any character. For example, all of the methods in Table 17-7 have overloaded forms that operate on **int**. Here is a sampling:

```
static boolean isDigit(int cp)
static boolean isLetter(int cp)
static int toLowerCase(int cp)
```

In addition to the methods overloaded to accept code points, **Character** adds methods that provide additional support for code points. A sampling is shown in Table 17-8.

Boolean

Boolean is a very thin wrapper around **boolean** values, which is useful mostly when you want to pass a **boolean** variable by reference. It contains the constants **TRUE** and **FALSE**, which define true and false **Boolean** objects. **Boolean** also defines the **TYPE** field, which is the **Class** object for **boolean**. **Boolean** defines these constructors:

```
Boolean(boolean boolValue)
Boolean(String boolString)
```

Method	Description
static int charCount(int <i>cp</i>)	Returns 1 if <i>cp</i> can be represented by a single char . It returns 2 if two chars are needed.
static int codePointAt(CharSequence <i>chars</i> , int <i>loc</i>)	Returns the code point at the location specified by <i>loc</i> .
static int codePointAt(char <i>chars</i> [], int <i>loc</i>)	Returns the code point at the location specified by <i>loc</i> .
static int codePointBefore(CharSequence <i>chars</i> , int <i>loc</i>)	Returns the code point at the location that precedes that specified by <i>loc</i> .
static int codePointBefore(char <i>chars</i> [], int <i>loc</i>)	Returns the code point at the location that precedes that specified by <i>loc</i> .
static boolean isBmpCodePoint(int <i>cp</i>)	Returns true if <i>cp</i> is part of the basic multilingual plane and false otherwise.
static boolean isHighSurrogate(char <i>ch</i>)	Returns true if <i>ch</i> contains a valid high surrogate character.
static boolean isLowSurrogate(char <i>ch</i>)	Returns true if <i>ch</i> contains a valid low surrogate character.
static boolean isSupplementaryCodePoint(int <i>cp</i>)	Returns true if <i>cp</i> contains a supplemental character.
static boolean isSurrogatePair(char <i>highCh</i> , char <i>lowCh</i>)	Returns true if <i>highCh</i> and <i>lowCh</i> form a valid surrogate pair.
static boolean isValidCodePoint(int <i>cp</i>)	Returns true if <i>cp</i> contains a valid code point.
static char[] toChars(int <i>cp</i>)	Converts the code point in <i>cp</i> into its char equivalent, which might require two chars . An array holding the result is returned.
static int toChars(int <i>cp</i> , char <i>target</i> [], int <i>loc</i>)	Converts the code point in <i>cp</i> into its char equivalent, storing the result in <i>target</i> , beginning at <i>loc</i> . Returns 1 if <i>cp</i> can be represented by a single char . It returns 2 otherwise.
static int toCodePoint(char <i>highCh</i> , char <i>lowCh</i>)	Converts <i>highCh</i> and <i>lowCh</i> into their equivalent code point.

Table 17-8 A Sampling of Methods That Provide Support for 32-Bit Unicode Code Points

In the first version, *boolValue* must be either **true** or **false**. In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be **true**. Otherwise, it will be **false**.

Boolean defines the methods shown in Table 17-9.

Method	Description
boolean booleanValue()	Returns boolean equivalent.
static int compare(boolean <i>b1</i> , boolean <i>b2</i>)	Returns zero if <i>b1</i> and <i>b2</i> contain the same value. Returns a positive value if <i>b1</i> is true and <i>b2</i> is false . Otherwise, returns a negative value.
int compareTo(Boolean <i>b</i>)	Returns zero if the invoking object and <i>b</i> contain the same value. Returns a positive value if the invoking object is true and <i>b</i> is false . Otherwise, returns a negative value.
boolean equals(Object <i>boolObj</i>)	Returns true if the invoking object is equivalent to <i>boolObj</i> . Otherwise, it returns false .
static Boolean getBoolean(String <i>propertyName</i>)	Returns true if the system property specified by <i>propertyName</i> is true . Otherwise, it returns false .
int hashCode()	Returns the hash code for the invoking object.
static int hashCode(boolean <i>boolVal</i>)	Returns the hash code for <i>boolVal</i> . (Added by JDK 8.)
static boolean logicalAnd(boolean <i>op1</i> , boolean <i>op2</i>)	Performs a logical AND of <i>op1</i> and <i>op2</i> and returns the result. (Added by JDK 8.)
static boolean logicalOr(boolean <i>op1</i> , boolean <i>op2</i>)	Performs a logical OR of <i>op1</i> and <i>op2</i> and returns the result. (Added by JDK 8.)
static boolean logicalXor(boolean <i>op1</i> , boolean <i>op2</i>)	Performs a logical XOR of <i>op1</i> and <i>op2</i> and returns the result. (Added by JDK 8.)
static boolean parseBoolean(String <i>str</i>)	Returns true if <i>str</i> contains the string "true". Case is not significant. Otherwise, returns false .
String toString()	Returns the string equivalent of the invoking object.
static String toString(boolean <i>boolVal</i>)	Returns the string equivalent of <i>boolVal</i> .
static Boolean valueOf(boolean <i>boolVal</i>)	Returns the Boolean equivalent of <i>boolVal</i> .
static Boolean valueOf(String <i>boolString</i>)	Returns true if <i>boolString</i> contains the string "true" (in uppercase or lowercase). Otherwise, it returns false .

Table 17-9 The Methods Defined by **Boolean**

Void

The **Void** class has one field, **TYPE**, which holds a reference to the **Class** object for type **void**. You do not create instances of this class.

Process

The abstract **Process** class encapsulates a *process*—that is, an executing program. It is used primarily as a superclass for the type of objects created by **exec()** in the **Runtime** class, or by **start()** in the **ProcessBuilder** class. **Process** contains the methods shown in Table 17-10.

Method	Description
void destroy()	Terminates the process.
Process destroyForcibly()	Forces termination of the invoking process. Returns a reference to the process. (Added by JDK 8.)
int exitValue()	Returns an exit code obtained from a subprocess.
InputStream getErrorStream()	Returns an input stream that reads input from the process' <code>err</code> output stream.
InputStream getInputStream()	Returns an input stream that reads input from the process' <code>out</code> output stream.
OutputStream getOutputStream()	Returns an output stream that writes output to the process' <code>in</code> input stream.
boolean isAlive()	Returns <code>true</code> if the invoking process is still active. Otherwise, returns <code>false</code> . (Added by JDK 8.)
int waitFor() throws InterruptedException	Returns the exit code returned by the process. This method does not return until the process on which it is called terminates.
boolean waitFor(long <i>waitTime</i> , TimeUnit <i>timeUnit</i>) throws InterruptedException	Waits for the invoking process to end. The amount of time to wait is specified by <i>waitTime</i> in the units specified by <i>timeUnit</i> . Returns <code>true</code> if the process has ended and <code>false</code> if the wait time runs out. (Added by JDK 8.)

Table 17-10 The Methods Defined by `Process`

Runtime

The `Runtime` class encapsulates the run-time environment. You cannot instantiate a `Runtime` object. However, you can get a reference to the current `Runtime` object by calling the static method `Runtime.getRuntime()`. Once you obtain a reference to the current `Runtime` object, you can call several methods that control the state and behavior of the Java Virtual Machine. Applets and other untrusted code typically cannot call any of the `Runtime` methods without raising a `SecurityException`. Several commonly used methods defined by `Runtime` are shown in Table 17-11.

Method	Description
void addShutdownHook(Thread <i>thrd</i>)	Registers <i>thrd</i> as a thread to be run when the Java Virtual Machine terminates.
Process exec(String <i>progName</i>) throws IOException	Executes the program specified by <i>progName</i> as a separate process. An object of type <code>Process</code> is returned that describes the new process.
Process exec(String <i>progName</i> , String <i>environment</i> []) throws IOException	Executes the program specified by <i>progName</i> as a separate process with the environment specified by <i>environment</i> . An object of type <code>Process</code> is returned that describes the new process.
Process exec(String <i>comLineArray</i> []) throws IOException	Executes the command line specified by the strings in <i>comLineArray</i> as a separate process. An object of type <code>Process</code> is returned that describes the new process.

Table 17-11 A Sampling of Methods Defined by `Runtime`

Method	Description
Process exec(String <i>comLineArray</i> [], String <i>environment</i> []) throws IOException	Executes the command line specified by the strings in <i>comLineArray</i> as a separate process with the environment specified by <i>environment</i> . An object of type Process is returned that describes the new process.
void exit(int <i>exitCode</i>)	Halts execution and returns the value of <i>exitCode</i> to the parent process. By convention, 0 indicates normal termination. All other values indicate some form of error.
long freeMemory()	Returns the approximate number of bytes of free memory available to the Java run-time system.
void gc()	Initiates garbage collection.
static Runtime getRuntime()	Returns the current Runtime object.
void halt(int <i>code</i>)	Immediately terminates the Java Virtual Machine. No termination threads or finalizers are run. The value of <i>code</i> is returned to the invoking process.
void load(String <i>libraryFileName</i>)	Loads the dynamic library whose file is specified by <i>libraryFileName</i> , which must specify its complete path.
void loadLibrary(String <i>libraryName</i>)	Loads the dynamic library whose name is associated with <i>libraryName</i> .
Boolean removeShutdownHook(Thread <i>thrd</i>)	Removes <i>thrd</i> from the list of threads to run when the Java Virtual Machine terminates. It returns true if successful—that is, if the thread was removed.
void runFinalization()	Initiates calls to the finalize() methods of unused but not yet recycled objects.
long totalMemory()	Returns the total number of bytes of memory available to the program.
void traceInstructions(boolean <i>traceOn</i>)	Turns on or off instruction tracing, depending upon the value of <i>traceOn</i> . If <i>traceOn</i> is true , the trace is displayed. If it is false , tracing is turned off.
void traceMethodCalls(boolean <i>traceOn</i>)	Turns on or off method call tracing, depending upon the value of <i>traceOn</i> . If <i>traceOn</i> is true , the trace is displayed. If it is false , tracing is turned off.

Table 17-11 A Sampling of Methods Defined by **Runtime** (*continued*)

Let's look at two of the most common uses of the **Runtime** class: memory management and executing additional processes.

Memory Management

Although Java provides automatic garbage collection, sometimes you will want to know how large the object heap is and how much of it is left. You can use this information, for example, to check your code for efficiency or to approximate how many more objects of a certain type can be instantiated. To obtain these values, use the **totalMemory()** and **freeMemory()** methods.

As mentioned in Part I, Java's garbage collector runs periodically to recycle unused objects. However, sometimes you will want to collect discarded objects prior to the collector's next appointed rounds. You can run the garbage collector on demand by calling the `gc()` method. A good thing to try is to call `gc()` and then call `freeMemory()` to get a baseline memory usage. Next, execute your code and call `freeMemory()` again to see how much memory it is allocating. The following program illustrates this idea:

```
// Demonstrate totalMemory(), freeMemory() and gc().  
  
class MemoryDemo {  
    public static void main(String args[]) {  
        Runtime r = Runtime.getRuntime();  
        long mem1, mem2;  
        Integer someints[] = new Integer[1000];  
  
        System.out.println("Total memory is: " +  
                           r.totalMemory());  
        mem1 = r.freeMemory();  
        System.out.println("Initial free memory: " + mem1);  
        r.gc();  
        mem1 = r.freeMemory();  
        System.out.println("Free memory after garbage collection: "  
                           + mem1);  
  
        for(int i=0; i<1000; i++)  
            someints[i] = new Integer(i); // allocate integers  
  
        mem2 = r.freeMemory();  
        System.out.println("Free memory after allocation: "  
                           + mem2);  
        System.out.println("Memory used by allocation: "  
                           + (mem1-mem2));  
  
        // discard Integers  
        for(int i=0; i<1000; i++) someints[i] = null;  
  
        r.gc(); // request garbage collection  
  
        mem2 = r.freeMemory();  
        System.out.println("Free memory after collecting" +  
                           " discarded Integers: " + mem2);  
    }  
}
```

Sample output from this program is shown here (of course, your actual results may vary):

```
Total memory is: 1048568  
Initial free memory: 751392  
Free memory after garbage collection: 841424  
Free memory after allocation: 824000  
Memory used by allocation: 17424  
Free memory after collecting discarded Integers: 842640
```

Executing Other Programs

In safe environments, you can use Java to execute other heavyweight processes (that is, programs) on your multitasking operating system. Several forms of the `exec()` method allow you to name the program you want to run as well as its input parameters. The `exec()` method returns a **Process** object, which can then be used to control how your Java program interacts with this new running process. Because Java can run on a variety of platforms and under a variety of operating systems, `exec()` is inherently environment-dependent.

The following example uses `exec()` to launch **notepad**, Windows' simple text editor. Obviously, this example must be run under the Windows operating system.

```
// Demonstrate exec().
class ExecDemo {
    public static void main(String args[]) {
        Runtime r = Runtime.getRuntime();
        Process p = null;

        try {
            p = r.exec("notepad");
        } catch (Exception e) {
            System.out.println("Error executing notepad.");
        }
    }
}
```

There are several alternative forms of `exec()`, but the one shown in the example is the most common. The **Process** object returned by `exec()` can be manipulated by **Process'** methods after the new program starts running. You can kill the subprocess with the `destroy()` method. The `waitFor()` method causes your program to wait until the subprocess finishes. The `exitValue()` method returns the value returned by the subprocess when it is finished. This is typically 0 if no problems occur. Here is the preceding `exec()` example modified to wait for the running process to exit:

```
// Wait until notepad is terminated.
class ExecDemoFini {
    public static void main(String args[]) {
        Runtime r = Runtime.getRuntime();
        Process p = null;

        try {
            p = r.exec("notepad");
            p.waitFor();
        } catch (Exception e) {
            System.out.println("Error executing notepad.");
        }
        System.out.println("Notepad returned " + p.exitValue());
    }
}
```

While a subprocess is running, you can write to and read from its standard input and output. The `getOutputStream()` and `getInputStream()` methods return the handles to standard **in** and **out** of the subprocess. (I/O is examined in detail in Chapter 20.)

ProcessBuilder

ProcessBuilder provides another way to start and manage processes (that is, programs). As explained earlier, all processes are represented by the **Process** class, and a process can be started by **Runtime.exec()**. **ProcessBuilder** offers more control over the processes. For example, you can set the current working directory.

ProcessBuilder defines these constructors:

```
ProcessBuilder(List<String> args)
ProcessBuilder(String ... args)
```

Here, *args* is a list of arguments that specify the name of the program to be executed along with any required command-line arguments. In the first constructor, the arguments are passed in a **List**. In the second, they are specified through a varargs parameter. Table 17-12 describes the methods defined by **ProcessBuilder**.

In Table 17-12, notice the methods that use the **ProcessBuilder.Redirect** class. This abstract class encapsulates an I/O source or target linked to a subprocess. Among other things, these methods enable you to redirect the source or target of I/O operations. For example, you can redirect to a file by calling **to()**, redirect from a file by calling **from()**, and append to a file by calling **appendTo()**. A **File** object linked to the file can be obtained by calling **file()**. These methods are shown here:

```
static ProcessBuilder.Redirect to(File f)
static ProcessBuilder.Redirect from(File f)
static ProcessBuilder.Redirect appendTo(File f)
File file()
```

Another method supported by **ProcessBuilder.Redirect** is **type()**, which returns a value of the enumeration type **ProcessBuilder.Redirect.Type**. This enumeration describes the type of the redirection. It defines these values: **APPEND**, **INHERIT**, **PIPE**, **READ**, or **WRITE**. **ProcessBuilder.Redirect** also defines the constants **INHERIT** and **PIPE**.

Method	Description
<code>List<String> command()</code>	Returns a reference to a List that contains the name of the program and its arguments. Changes to this list affect the invoking object.
<code>ProcessBuilder command(List<String> args)</code>	Sets the name of the program and its arguments to those specified by <i>args</i> . Changes to this list affect the invoking object. Returns a reference to the invoking object.
<code>ProcessBuilder command(String ... args)</code>	Sets the name of the program and its arguments to those specified by <i>args</i> . Returns a reference to the invoking object.
<code>File directory()</code>	Returns the current working directory of the invoking object. This value will be null if the directory is the same as that of the Java program that started the process.

Table 17-12 The Methods Defined by **ProcessBuilder**

Method	Description
ProcessBuilder directory(File <i>dir</i>)	Sets the current working directory of the invoking object. Returns a reference to the invoking object.
Map<String, String> environment()	Returns the environmental variables associated with the invoking object as key/value pairs.
ProcessBuilder inheritIO()	Causes the invoked process to use the same source and target for the standard I/O streams as the invoking process.
ProcessBuilder.Redirect redirectError()	Returns the target for standard error as a ProcessBuilder.Redirect object.
ProcessBuilder redirectError(File <i>f</i>)	Sets the target for standard error to the specified file. Returns a reference to the invoking object.
ProcessBuilder redirectError(ProcessBuilder.Redirect <i>target</i>)	Sets the target for standard error as specified by <i>target</i> . Returns a reference to the invoking object.
boolean redirectErrorStream()	Returns true if the standard error stream has been redirected to the standard output stream. Returns false if the streams are separate.
ProcessBuilder redirectErrorStream(boolean <i>merge</i>)	If <i>merge</i> is true , then the standard error stream is redirected to standard output. If <i>merge</i> is false , the streams are separated, which is the default state. Returns a reference to the invoking object.
ProcessBuilder.Redirect redirectInput()	Returns the source for standard input as a ProcessBuilder.Redirect object.
ProcessBuilder redirectInput(File <i>f</i>)	Sets the source for standard input to the specified file. Returns a reference to the invoking object.
ProcessBuilder redirectInput(ProcessBuilder.Redirect <i>source</i>)	Sets the source for standard input as specified by <i>source</i> . Returns a reference to the invoking object.
ProcessBuilder.Redirect redirectOutput()	Returns the target for standard output as a ProcessBuilder.Redirect object.
ProcessBuilder redirectOutput(File <i>f</i>)	Sets the target for standard output to the specified file. Returns a reference to the invoking object.
ProcessBuilder redirectOutput(ProcessBuilder.Redirect <i>target</i>)	Sets the target for standard output as specified by <i>target</i> . Returns a reference to the invoking object.
Process start() throws IOException	Begins the process specified by the invoking object. In other words, it runs the specified program.

Table 17-12 The Methods Defined by **ProcessBuilder** (*continued*)

To create a process using **ProcessBuilder**, simply create an instance of **ProcessBuilder**, specifying the name of the program and any needed arguments. To begin execution of the program, call **start()** on that instance. Here is an example that executes the Windows text editor **notepad**. Notice that it specifies the name of the file to edit as an argument.

```
class PBDemo {
    public static void main(String args[]) {

        try {
            ProcessBuilder proc =
                new ProcessBuilder("notepad.exe", "testfile");
            proc.start();
        } catch (Exception e) {
            System.out.println("Error executing notepad.");
        }
    }
}
```

System

The **System** class holds a collection of static methods and variables. The standard input, output, and error output of the Java run time are stored in the **in**, **out**, and **err** variables. The methods defined by **System** are shown in Table 17-13. Many of the methods throw a **SecurityException** if the operation is not permitted by the security manager.

Let's look at some common uses of **System**.

Method	Description
static void arraycopy(Object <i>source</i> , int <i>sourceStart</i> , Object <i>target</i> , int <i>targetStart</i> , int <i>size</i>)	Copies an array. The array to be copied is passed in <i>source</i> , and the index at which point the copy will begin within <i>source</i> is passed in <i>sourceStart</i> . The array that will receive the copy is passed in <i>target</i> , and the index at which point the copy will begin within <i>target</i> is passed in <i>targetStart</i> . <i>size</i> is the number of elements that are copied.
static String clearProperty(String <i>which</i>)	Deletes the environmental variable specified by <i>which</i> . The previous value associated with <i>which</i> is returned.
static Console console()	Returns the console associated with the JVM. null is returned if the JVM currently has no console.
static long currentTimeMillis()	Returns the current time in terms of milliseconds since midnight, January 1, 1970.
static void exit(int <i>exitCode</i>)	Halts execution and returns the value of <i>exitCode</i> to the parent process (usually the operating system). By convention, 0 indicates normal termination. All other values indicate some form of error.
static void gc()	Initiates garbage collection.

Table 17-13 The Methods Defined by **System**

Method	Description
static Map<String, String> getenv()	Returns a Map that contains the current environmental variables and their values.
static String getenv(String <i>which</i>)	Returns the value associated with the environmental variable passed in <i>which</i> .
static Properties getProperties()	Returns the properties associated with the Java runtime system. (The Properties class is described in Chapter 18.)
static String getProperty(String <i>which</i>)	Returns the property associated with <i>which</i> . A null object is returned if the desired property is not found.
static String getProperty(String <i>which</i> , String <i>default</i>)	Returns the property associated with <i>which</i> . If the desired property is not found, <i>default</i> is returned.
static SecurityManager getSecurityManager()	Returns the current security manager or a null object if no security manager is installed.
static int identityHashCode(Object <i>obj</i>)	Returns the identity hash code for <i>obj</i> .
static Channel inheritedChannel() throws IOException	Returns the channel inherited by the Java Virtual Machine. Returns null if no channel is inherited.
static String lineSeparator()	Returns a string that contains the line-separator characters.
static void load(String <i>libraryFileName</i>)	Loads the dynamic library whose file is specified by <i>libraryFileName</i> , which must specify its complete path.
static void loadLibrary(String <i>libraryName</i>)	Loads the dynamic library whose name is associated with <i>libraryName</i> .
static String mapLibraryName(String <i>lib</i>)	Returns a platform-specific name for the library named <i>lib</i> .
static long nanoTime()	Obtains the most precise timer in the system and returns its value in terms of nanoseconds since some arbitrary starting point. The accuracy of the timer is unknowable.
static void runFinalization()	Initiates calls to the finalize() methods of unused but not yet recycled objects.
static void setErr(PrintStream <i>eStream</i>)	Sets the standard err stream to <i>eStream</i> .
static void setIn(InputStream <i>iStream</i>)	Sets the standard in stream to <i>iStream</i> .
static void setOut(PrintStream <i>oStream</i>)	Sets the standard out stream to <i>oStream</i> .
static void setProperties(Properties <i>sysProperties</i>)	Sets the current system properties as specified by <i>sysProperties</i> .
static String setProperty(String <i>which</i> , String <i>v</i>)	Assigns the value <i>v</i> to the property named <i>which</i> .
static void setSecurityManager(SecurityManager <i>secMan</i>)	Sets the security manager to that specified by <i>secMan</i> .

Table 17-13 The Methods Defined by **System** (*continued*)

Using currentTimeMillis() to Time Program Execution

One use of the **System** class that you might find particularly interesting is to use the **currentTimeMillis()** method to time how long various parts of your program take to execute. The **currentTimeMillis()** method returns the current time in terms of milliseconds since midnight, January 1, 1970. To time a section of your program, store this value just before beginning the section in question. Immediately upon completion, call **currentTimeMillis()** again. The elapsed time will be the ending time minus the starting time. The following program demonstrates this:

```
// Timing program execution.

class Elapsed {
    public static void main(String args[]) {
        long start, end;

        System.out.println("Timing a for loop from 0 to 100,000,000");

        // time a for loop from 0 to 100,000,000

        start = System.currentTimeMillis(); // get starting time
        for(long i=0; i < 100000000L; i++) ;
        end = System.currentTimeMillis(); // get ending time

        System.out.println("Elapsed time: " + (end-start));
    }
}
```

Here is a sample run (remember that your results probably will differ):

```
Timing a for loop from 0 to 100,000,000
Elapsed time: 10
```

If your system has a timer that offers nanosecond precision, then you could rewrite the preceding program to use **nanoTime()** rather than **currentTimeMillis()**. For example, here is the key portion of the program rewritten to use **nanoTime()**:

```
start = System.nanoTime(); // get starting time
for(long i=0; i < 100000000L; i++) ;
end = System.nanoTime(); // get ending time
```

Using arraycopy()

The **arraycopy()** method can be used to copy quickly an array of any type from one place to another. This is much faster than the equivalent loop written out longhand in Java. Here is an example of two arrays being copied by the **arraycopy()** method. First, **a** is copied to **b**. Next, all of **a**'s elements are shifted *down* by one. Then, **b** is shifted *up* by one.

```
// Using arraycopy().

class ACDemo {
    static byte a[] = { 65, 66, 67, 68, 69, 70, 71, 72, 73, 74 };
    static byte b[] = { 77, 77, 77, 77, 77, 77, 77, 77, 77, 77 };
```

```

public static void main(String args[]) {
    System.out.println("a = " + new String(a));
    System.out.println("b = " + new String(b));
    System.arraycopy(a, 0, b, 0, a.length);
    System.out.println("a = " + new String(a));
    System.out.println("b = " + new String(b));
    System.arraycopy(a, 0, a, 1, a.length - 1);
    System.arraycopy(b, 1, b, 0, b.length - 1);
    System.out.println("a = " + new String(a));
    System.out.println("b = " + new String(b));
}
}

```

As you can see from the following output, you can copy using the same source and destination in either direction:

```

a = ABCDEFGHIJ
b = MMMMMMMMM
a = ABCDEFGHIJ
b = ABCDEFGHIJ
a = AABCD EFGHI
b = BCDEFGHIJJ

```

Environment Properties

The following properties are available in all cases:

file.separator	java.specification.version	java.vm.version
java.class.path	java.vendor	line.separator
java.class.version	java.vendor.url	os.arch
java.compiler	java.version	os.name
java.ext.dirs	java.vm.name	os.version
java.home	java.vm.specification.name	path.separator
java.io.tmpdir	java.vm.specification.vendor	user.dir
java.library.path	java.vm.specification.version	user.home
java.specification.name	java.vm.vendor	user.name
java.specification.vendor		

You can obtain the values of various environment variables by calling the **System.getProperty()** method. For example, the following program displays the path to the current user directory:

```

class ShowUserDir {
    public static void main(String args[]) {
        System.out.println(System.getProperty("user.dir"));
    }
}

```

Object

As mentioned in Part I, **Object** is a superclass of all other classes. **Object** defines the methods shown in Table 17-14, which are available to every object.

Using `clone()` and the **Cloneable** Interface

Most of the methods defined by **Object** are discussed elsewhere in this book. However, one deserves special attention: `clone()`. The `clone()` method generates a duplicate copy of the object on which it is called. Only classes that implement the **Cloneable** interface can be cloned.

The **Cloneable** interface defines no members. It is used to indicate that a class allows a bitwise copy of an object (that is, a *clone*) to be made. If you try to call `clone()` on a class that does not implement **Cloneable**, a **CloneNotSupportedException** is thrown. When a clone is made, the constructor for the object being cloned is *not* called. As implemented by **Object**, a clone is simply an exact copy of the original.

Cloning is a potentially dangerous action, because it can cause unintended side effects. For example, if the object being cloned contains a reference variable called *obRef*, then when the clone is made, *obRef* in the clone will refer to the same object as does *obRef* in the

Method	Description
<code>Object clone() throws CloneNotSupportedException</code>	Creates a new object that is the same as the invoking object.
<code>boolean equals(Object object)</code>	Returns true if the invoking object is equivalent to <i>object</i> .
<code>void finalize() throws Throwable</code>	Default <code>finalize()</code> method. It is called before an unused object is recycled.
<code>final Class<?> getClass()</code>	Obtains a Class object that describes the invoking object.
<code>int hashCode()</code>	Returns the hash code associated with the invoking object.
<code>final void notify()</code>	Resumes execution of a thread waiting on the invoking object.
<code>final void notifyAll()</code>	Resumes execution of all threads waiting on the invoking object.
<code>String toString()</code>	Returns a string that describes the object.
<code>final void wait() throws InterruptedException</code>	Waits on another thread of execution.
<code>final void wait(long milliseconds) throws InterruptedException</code>	Waits up to the specified number of <i>milliseconds</i> on another thread of execution.
<code>final void wait(long milliseconds, int nanoseconds) throws InterruptedException</code>	Waits up to the specified number of <i>milliseconds</i> plus <i>nanoseconds</i> on another thread of execution.

Table 17-14 The Methods Defined by **Object**

original. If the clone makes a change to the contents of the object referred to by *obRef*, then it will be changed for the original object, too. Here is another example: If an object opens an I/O stream and is then cloned, two objects will be capable of operating on the same stream. Further, if one of these objects closes the stream, the other object might still attempt to write to it, causing an error. In some cases, you will need to override the **clone()** method defined by **Object** to handle these types of problems.

Because cloning can cause problems, **clone()** is declared as **protected** inside **Object**. This means that it must either be called from within a method defined by the class that implements **Cloneable**, or it must be explicitly overridden by that class so that it is public. Let's look at an example of each approach.

The following program implements **Cloneable** and defines the method **cloneTest()**, which calls **clone()** in **Object**:

```
// Demonstrate the clone() method

class TestClone implements Cloneable {
    int a;
    double b;

    // This method calls Object's clone().
    TestClone cloneTest() {
        try {
            // call clone in Object.
            return (TestClone) super.clone();
        } catch(CloneNotSupportedException e) {
            System.out.println("Cloning not allowed.");
            return this;
        }
    }
}

class CloneDemo {
    public static void main(String args[]) {
        TestClone x1 = new TestClone();
        TestClone x2;

        x1.a = 10;
        x1.b = 20.98;

        x2 = x1.cloneTest(); // clone x1

        System.out.println("x1: " + x1.a + " " + x1.b);
        System.out.println("x2: " + x2.a + " " + x2.b);
    }
}
```

Here, the method **cloneTest()** calls **clone()** in **Object** and returns the result. Notice that the object returned by **clone()** must be cast into its appropriate type (**TestClone**).

The following example overrides `clone()` so that it can be called from code outside of its class. To do this, its access specifier must be **public**, as shown here:

```
// Override the clone() method.

class TestClone implements Cloneable {
    int a;
    double b;

    // clone() is now overridden and is public.
    public Object clone() {
        try {
            // call clone in Object.
            return super.clone();
        } catch(CloneNotSupportedException e) {
            System.out.println("Cloning not allowed.");
            return this;
        }
    }

    class CloneDemo2 {
        public static void main(String args[]) {
            TestClone x1 = new TestClone();
            TestClone x2;

            x1.a = 10;
            x1.b = 20.98;

            // here, clone() is called directly.
            x2 = (TestClone) x1.clone();

            System.out.println("x1: " + x1.a + " " + x1.b);
            System.out.println("x2: " + x2.a + " " + x2.b);
        }
    }
}
```

The side effects caused by cloning are sometimes difficult to see at first. It is easy to think that a class is safe for cloning when it actually is not. In general, you should not implement **Cloneable** for any class without good reason.

Class

Class encapsulates the run-time state of a class or interface. Objects of type **Class** are created automatically, when classes are loaded. You cannot explicitly declare a **Class** object. Generally, you obtain a **Class** object by calling the `getClass()` method defined by **Object**. **Class** is a generic type that is declared as shown here:

```
class Class<T>
```

Here, **T** is the type of the class or interface represented. A sampling of methods defined by **Class** is shown in Table 17-15.

Method	Description
static Class<?> forName(String <i>name</i>) throws ClassNotFoundException	Returns a Class object given its complete name.
static Class<?> forName(String <i>name</i> , boolean <i>how</i> , ClassLoader <i>ldr</i>) throws ClassNotFoundException	Returns a Class object given its complete name. The object is loaded using the loader specified by <i>ldr</i> . If <i>how</i> is true , the object is initialized; otherwise, it is not.
<A extends Annotation> A getAnnotation(Class<A> <i>annoType</i>)	Returns an Annotation object that contains the annotation associated with <i>annoType</i> for the invoking object.
Annotation[] getAnnotations()	Obtains all annotations associated with the invoking object and stores them in an array of Annotation objects. Returns a reference to this array.
<A extends Annotation> A[] getAnnotationsByType(Class<A> <i>annoType</i>)	Returns an array of the annotations (including repeated annotations) of <i>annoType</i> associated with the invoking object. (Added by JDK 8.)
Class<?>[] getClasses()	Returns a Class object for each public class and interface that is a member of the class represented by the invoking object.
ClassLoader getClassLoader()	Returns the ClassLoader object that loaded the class or interface.
Constructor<T> getConstructor(Class<?> ... <i>paramTypes</i>) throws NoSuchMethodException, SecurityException	Returns a Constructor object that represents the constructor for the class represented by the invoking object that has the parameter types specified by <i>paramTypes</i> .
Constructor<?>[] getConstructors() throws SecurityException	Obtains a Constructor object for each public constructor of the class represented by the invoking object and stores them in an array. Returns a reference to this array.
Annotation[] getDeclaredAnnotations()	Obtains an Annotation object for all the annotations that are declared by the invoking object and stores them in an array. Returns a reference to this array. (Inherited annotations are ignored.)
<A extends Annotation> A[] getDeclaredAnnotationsByType(Class<A> <i>annoType</i>)	Returns an array of the non-inherited annotations (including repeated annotations) of <i>annoType</i> associated with the invoking object. (Added by JDK 8.)
Constructor<?>[] getDeclaredConstructors() throws SecurityException	Obtains a Constructor object for each constructor declared by the class represented by the invoking object and stores them in an array. Returns a reference to this array. (Superclass constructors are ignored.)
Field[] getDeclaredFields() throws SecurityException	Obtains a Field object for each field declared by the class or interface represented by the invoking object and stores them in an array. Returns a reference to this array. (Inherited fields are ignored.)

Table 17-15 A Sampling of Methods Defined by **Class**

Method	Description
Method[] getDeclaredMethods() throws SecurityException	Obtains a Method object for each method declared by the class or interface represented by the invoking object and stores them in an array. Returns a reference to this array. (Inherited methods are ignored.)
Field getField(String <i>fieldName</i>) throws NoSuchMethodException, SecurityException	Returns a Field object that represents the public field specified by <i>fieldName</i> for the class or interface represented by the invoking object.
Field[] getFields() throws SecurityException	Obtains a Field object for each public field of the class or interface represented by the invoking object and stores them in an array. Returns a reference to this array.
Class<?>[] getInterfaces()	When invoked on an object that represents a class, this method returns an array of the interfaces implemented by that class. When invoked on an object that represents an interface, this method returns an array of interfaces extended by that interface.
Method getMethod(String <i>methName</i> , Class<?> ... <i>paramTypes</i>) throws NoSuchMethodException, SecurityException	Returns a Method object that represents the public method specified by <i>methName</i> and having the parameter types specified by <i>paramTypes</i> in the class or interface represented by the invoking object.
Method[] getMethods() throws SecurityException	Obtains a Method object for each public method of the class or interface represented by the invoking object and stores them in an array. Returns a reference to this array.
String getName()	Returns the complete name of the class or interface of the type represented by the invoking object.
ProtectionDomain getProtectionDomain()	Returns the protection domain associated with the invoking object.
Class<? super T> getSuperclass()	Returns the superclass of the type represented by the invoking object. The return value is null if the represented type is Object or not a class.
boolean isInterface()	Returns true if the type represented by the invoking object is an interface. Otherwise, it returns false .
T newInstance() throws IllegalAccessException, InstantiationException	Creates a new instance (i.e., a new object) that is of the same type as that represented by invoking object. This is equivalent to using new with the class' default constructor. The new object is returned. This method will fail if the represented type is abstract, not a class, or does not have a default constructor.
String toString()	Returns the string representation of the type represented by the invoking object or interface.

Table 17-15 A Sampling of Methods Defined by **Class** (continued)

The methods defined by **Class** are often useful in situations where run-time type information about an object is required. As Table 17-15 shows, methods are provided that allow you to determine additional information about a particular class, such as its public constructors, fields, and methods. Among other things, this is important for the Java Beans functionality, which is discussed later in this book.

The following program demonstrates **getClass()** (inherited from **Object**) and **getSuperclass()** (from **Class**):

```
// Demonstrate Run-Time Type Information.

class X {
    int a;
    float b;
}

class Y extends X {
    double c;
}

class RTTI {
    public static void main(String args[]) {
        X x = new X();
        Y y = new Y();
        Class<?> clObj;

        clObj = x.getClass(); // get Class reference
        System.out.println("x is object of type: " +
                           clObj.getName());

        clObj = y.getClass(); // get Class reference
        System.out.println("y is object of type: " +
                           clObj.getName());
        clObj = clObj.getSuperclass();
        System.out.println("y's superclass is " +
                           clObj.getName());
    }
}
```

The output from this program is shown here:

```
x is object of type: X
y is object of type: Y
y's superclass is X
```

ClassLoader

The abstract class **ClassLoader** defines how classes are loaded. Your application can create subclasses that extend **ClassLoader**, implementing its methods. Doing so allows you to load classes in some way other than the way they are normally loaded by the Java run-time system. However, this is not something that you will normally need to do.

Math

The **Math** class contains all the floating-point functions that are used for geometry and trigonometry, as well as several general-purpose methods. **Math** defines two **double** constants: **E** (approximately 2.72) and **PI** (approximately 3.14).

Trigonometric Functions

The following methods accept a **double** parameter for an angle in radians and return the result of their respective trigonometric function:

Method	Description
static double sin(double <i>arg</i>)	Returns the sine of the angle specified by <i>arg</i> in radians.
static double cos(double <i>arg</i>)	Returns the cosine of the angle specified by <i>arg</i> in radians.
static double tan(double <i>arg</i>)	Returns the tangent of the angle specified by <i>arg</i> in radians.

The next methods take as a parameter the result of a trigonometric function and return, in radians, the angle that would produce that result. They are the inverse of their non-arc companions.

Method	Description
static double asin(double <i>arg</i>)	Returns the angle whose sine is specified by <i>arg</i> .
static double acos(double <i>arg</i>)	Returns the angle whose cosine is specified by <i>arg</i> .
static double atan(double <i>arg</i>)	Returns the angle whose tangent is specified by <i>arg</i> .
static double atan2(double <i>x</i> , double <i>y</i>)	Returns the angle whose tangent is <i>x/y</i> .

The next methods compute the hyperbolic sine, cosine, and tangent of an angle:

Method	Description
static double sinh(double <i>arg</i>)	Returns the hyperbolic sine of the angle specified by <i>arg</i> .
static double cosh(double <i>arg</i>)	Returns the hyperbolic cosine of the angle specified by <i>arg</i> .
static double tanh(double <i>arg</i>)	Returns the hyperbolic tangent of the angle specified by <i>arg</i> .

Exponential Functions

Math defines the following exponential methods:

Method	Description
static double cbrt(double <i>arg</i>)	Returns the cube root of <i>arg</i> .
static double exp(double <i>arg</i>)	Returns e to the <i>arg</i> .
static double expm1(double <i>arg</i>)	Returns e to the <i>arg</i> -1.
static double log(double <i>arg</i>)	Returns the natural logarithm of <i>arg</i> .
static double log10(double <i>arg</i>)	Returns the base 10 logarithm for <i>arg</i> .
static double log1p(double <i>arg</i>)	Returns the natural logarithm for <i>arg</i> +1.
static double pow(double <i>y</i> , double <i>x</i>)	Returns <i>y</i> raised to the <i>x</i> ; for example, pow(2.0, 3.0) returns 8.0.
static double scalb(double <i>arg</i> , int <i>factor</i>)	Returns <i>arg</i> × 2^{factor} .
static float scalb(float <i>arg</i> , int <i>factor</i>)	Returns <i>arg</i> × 2^{factor} .
static double sqrt(double <i>arg</i>)	Returns the square root of <i>arg</i> .

Rounding Functions

The **Math** class defines several methods that provide various types of rounding operations. They are shown in Table 17-16. Notice the two **ulp()** methods at the end of the table. In this context, *ulp* stands for *units in the last place*. It indicates the distance between a value and the next higher value. It can be used to help assess the accuracy of a result.

Method	Description
static int abs(int <i>arg</i>)	Returns the absolute value of <i>arg</i> .
static long abs(long <i>arg</i>)	Returns the absolute value of <i>arg</i> .
static float abs(float <i>arg</i>)	Returns the absolute value of <i>arg</i> .
static double abs(double <i>arg</i>)	Returns the absolute value of <i>arg</i> .
static double ceil(double <i>arg</i>)	Returns the smallest whole number greater than or equal to <i>arg</i> .
static double floor(double <i>arg</i>)	Returns the largest whole number less than or equal to <i>arg</i> .
static int floorDiv(int <i>dividend</i> , int <i>divisor</i>)	Returns the floor of the result of <i>dividend</i> / <i>divisor</i> . (Added by JDK 8.)
static long floorDiv(long <i>dividend</i> , long <i>divisor</i>)	Returns the floor of the result of <i>dividend</i> / <i>divisor</i> . (Added by JDK 8.)
static int floorMod(int <i>dividend</i> , int <i>divisor</i>)	Returns the floor of the remainder of <i>dividend</i> / <i>divisor</i> . (Added by JDK 8.)
static long floorMod(long <i>dividend</i> , long <i>divisor</i>)	Returns the floor of the remainder of <i>dividend</i> / <i>divisor</i> . (Added by JDK 8.)

Table 17-16 The Rounding Methods Defined by **Math**

Method	Description
static int max(int <i>x</i> , int <i>y</i>)	Returns the maximum of <i>x</i> and <i>y</i> .
static long max(long <i>x</i> , long <i>y</i>)	Returns the maximum of <i>x</i> and <i>y</i> .
static float max(float <i>x</i> , float <i>y</i>)	Returns the maximum of <i>x</i> and <i>y</i> .
static double max(double <i>x</i> , double <i>y</i>)	Returns the maximum of <i>x</i> and <i>y</i> .
static int min(int <i>x</i> , int <i>y</i>)	Returns the minimum of <i>x</i> and <i>y</i> .
static long min(long <i>x</i> , long <i>y</i>)	Returns the minimum of <i>x</i> and <i>y</i> .
static float min(float <i>x</i> , float <i>y</i>)	Returns the minimum of <i>x</i> and <i>y</i> .
static double min(double <i>x</i> , double <i>y</i>)	Returns the minimum of <i>x</i> and <i>y</i> .
static double nextAfter(double <i>arg</i> , double <i>toward</i>)	Beginning with the value of <i>arg</i> , returns the next value in the direction of <i>toward</i> . If <i>arg</i> == <i>toward</i> , then <i>toward</i> is returned.
static float nextAfter(float <i>arg</i> , double <i>toward</i>)	Beginning with the value of <i>arg</i> , returns the next value in the direction of <i>toward</i> . If <i>arg</i> == <i>toward</i> , then <i>toward</i> is returned.
static double nextDown(double <i>val</i>)	Returns the next value lower than <i>val</i> . (Added by JDK 8.)
static float nextDown(float <i>val</i>)	Returns the next value lower than <i>val</i> . (Added by JDK 8.)
static double nextUp(double <i>arg</i>)	Returns the next value in the positive direction from <i>arg</i> .
static float nextUp(float <i>arg</i>)	Returns the next value in the positive direction from <i>arg</i> .
static double rint(double <i>arg</i>)	Returns the integer nearest in value to <i>arg</i> .
static int round(float <i>arg</i>)	Returns <i>arg</i> rounded up to the nearest int .
static long round(double <i>arg</i>)	Returns <i>arg</i> rounded up to the nearest long .
static float ulp(float <i>arg</i>)	Returns the ulp for <i>arg</i> .
static double ulp(double <i>arg</i>)	Returns the ulp for <i>arg</i> .

Table 17-16 The Rounding Methods Defined by **Math** (continued)

Miscellaneous Math Methods

In addition to the methods just shown, **Math** defines several other methods, which are shown in Table 17-17. Notice that several of the methods use the suffix **Exact**. These were added by JDK 8. They throw an **ArithmaticException** if overflow occurs. Thus, these methods give you an easy way to watch various operations for overflow.

Method	Description
static int addExact(int <i>arg1</i> , int <i>arg2</i>)	Returns <i>arg1</i> + <i>arg2</i> . Throws an ArithmeticException if overflow occurs. (Added by JDK 8.)
static long addExact(long <i>arg1</i> , long <i>arg2</i>)	Returns <i>arg1</i> + <i>arg2</i> . Throws an ArithmeticException if overflow occurs. (Added by JDK 8.)
static double copySign(double <i>arg</i> , double <i>signarg</i>)	Returns <i>arg</i> with same sign as that of <i>signarg</i> .
static float copySign(float <i>arg</i> , float <i>signarg</i>)	Returns <i>arg</i> with same sign as that of <i>signarg</i> .
static int decrementExact(int <i>arg</i>)	Returns <i>arg</i> - 1. Throws an ArithmeticException if overflow occurs. (Added by JDK 8.)
static long decrementExact(long <i>arg</i>)	Returns <i>arg</i> - 1. Throws an ArithmeticException if overflow occurs. (Added by JDK 8.)
static int getExponent(double <i>arg</i>)	Returns the base-2 exponent used by the binary representation of <i>arg</i> .
static int getExponent(float <i>arg</i>)	Returns the base-2 exponent used by the binary representation of <i>arg</i> .
static hypot(double <i>side1</i> , double <i>side2</i>)	Returns the length of the hypotenuse of a right triangle given the length of the two opposing sides.
static double IEEEremainder(double <i>dividend</i> , double <i>divisor</i>)	Returns the remainder of <i>dividend</i> / <i>divisor</i> .
static int incrementExact(int <i>arg</i>)	Returns <i>arg</i> + 1. Throws an ArithmeticException if overflow occurs. (Added by JDK 8.)
static long incrementExact(long <i>arg</i>)	Returns <i>arg</i> + 1. Throws an ArithmeticException if overflow occurs. (Added by JDK 8.)
static int multiplyExact(int <i>arg1</i> , int <i>arg2</i>)	Returns <i>arg1</i> * <i>arg2</i> . Throws an ArithmeticException if overflow occurs. (Added by JDK 8.)
static long multiplyExact(long <i>arg1</i> , long <i>arg2</i>)	Returns <i>arg1</i> * <i>arg2</i> . Throws an ArithmeticException if overflow occurs. (Added by JDK 8.)
static int negateExact(int <i>arg</i>)	Returns - <i>arg</i> . Throws an ArithmeticException if overflow occurs. (Added by JDK 8.)
static long negateExact(long <i>arg</i>)	Returns - <i>arg</i> . Throws an ArithmeticException if overflow occurs. (Added by JDK 8.)
static double random()	Returns a pseudorandom number between 0 and 1.
static float signum(double <i>arg</i>)	Determines the sign of a value. It returns 0 if <i>arg</i> is 0, 1 if <i>arg</i> is greater than 0, and -1 if <i>arg</i> is less than 0.
static float signum(float <i>arg</i>)	Determines the sign of a value. It returns 0 if <i>arg</i> is 0, 1 if <i>arg</i> is greater than 0, and -1 if <i>arg</i> is less than 0.
static int subtractExact(int <i>arg1</i> , int <i>arg2</i>)	Returns <i>arg1</i> - <i>arg2</i> . Throws an ArithmeticException if overflow occurs. (Added by JDK 8.)
static long subtractExact(long <i>arg1</i> , long <i>arg2</i>)	Returns <i>arg1</i> - <i>arg2</i> . Throws an ArithmeticException if overflow occurs. (Added by JDK 8.)
static double toDegrees(double <i>angle</i>)	Converts radians to degrees. The angle passed to <i>angle</i> must be specified in radians. The result in degrees is returned.
static int toIntExact(long <i>arg</i>)	Returns <i>arg</i> as an int. Throws an ArithmeticException if overflow occurs. (Added by JDK 8.)
static double toRadians(double <i>angle</i>)	Converts degrees to radians. The <i>angle</i> passed to <i>angle</i> must be specified in degrees. The result in radians is returned.

Table 17-17 Other Methods Defined by **Math**

The following program demonstrates `toRadians()` and `toDegrees()`:

```
// Demonstrate toDegrees() and toRadians().
class Angles {
    public static void main(String args[]) {
        double theta = 120.0;

        System.out.println(theta + " degrees is " +
                           Math.toRadians(theta) + " radians.");
        theta = 1.312;
        System.out.println(theta + " radians is " +
                           Math.toDegrees(theta) + " degrees.");
    }
}
```

The output is shown here:

```
120.0 degrees is 2.0943951023931953 radians.
1.312 radians is 75.17206272116401 degrees.
```

StrictMath

The **StrictMath** class defines a complete set of mathematical methods that parallel those in **Math**. The difference is that the **StrictMath** version is guaranteed to generate precisely identical results across all Java implementations, whereas the methods in **Math** are given more latitude in order to improve performance.

Compiler

The **Compiler** class supports the creation of Java environments in which Java bytecode is compiled into executable code rather than interpreted. It is not for normal programming use.

Thread, ThreadGroup, and Runnable

The **Runnable** interface and the **Thread** and **ThreadGroup** classes support multithreaded programming. Each is examined next.

NOTE An overview of the techniques used to manage threads, implement the **Runnable** interface, and create multithreaded programs is presented in Chapter 11.

The Runnable Interface

The **Runnable** interface must be implemented by any class that will initiate a separate thread of execution. **Runnable** only defines one abstract method, called `run()`, which is the entry point to the thread. It is defined like this:

```
void run()
```

Threads that you create must implement this method.

Thread

Thread creates a new thread of execution. It implements **Runnable** and defines the following commonly used constructors:

```
Thread()
Thread(Runnable threadOb)
Thread(Runnable threadOb, String threadName)
Thread(String threadName)
Thread(ThreadGroup groupOb, Runnable threadOb)
Thread(ThreadGroup groupOb, Runnable threadOb, String threadName)
Thread(ThreadGroup groupOb, String threadName)
```

threadOb is an instance of a class that implements the **Runnable** interface and defines where execution of the thread will begin. The name of the thread is specified by *threadName*. When a name is not specified, one is created by the Java Virtual Machine. *groupOb* specifies the thread group to which the new thread will belong. When no thread group is specified, the new thread belongs to the same group as the parent thread.

The following constants are defined by **Thread**:

```
MAX_PRIORITY
MIN_PRIORITY
NORM_PRIORITY
```

As expected, these constants specify the maximum, minimum, and default thread priorities.

The methods defined by **Thread** are shown in Table 17-18. In early versions of Java, **Thread** also included the methods **stop()**, **suspend()**, and **resume()**. However, as explained in Chapter 11, these were deprecated because they were inherently unstable. Also deprecated are **countStackFrames()**, because it calls **suspend()**, and **destroy()**, because it can cause deadlock.

Method	Description
static int activeCount()	Returns the approximate number of active threads in the group to which the thread belongs.
final void checkAccess()	Causes the security manager to verify that the current thread can access and/or change the thread on which checkAccess() is called.
static Thread currentThread()	Returns a Thread object that encapsulates the thread that calls this method.
static void dumpStack()	Displays the call stack for the thread.
static int enumerate(Thread <i>threads</i> [])	Puts copies of all Thread objects in the current thread's group into <i>threads</i> . The number of threads is returned.
static Map<Thread, StackTraceElement[]> getAllStackTraces()	Returns a Map that contains the stack traces for all active threads. In the map, each entry consists of a key, which is the Thread object, and its value, which is an array of StackTraceElement .

Table 17-18 The Methods Defined by **Thread**

Method	Description
ClassLoader getContextClassLoader()	Returns the context class loader that is used to load classes and resources for this thread.
static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()	Returns the default uncaught exception handler.
long getID()	Returns the ID of the invoking thread.
final String getName()	Returns the thread's name.
final int getPriority()	Returns the thread's priority setting.
StackTraceElement[] getStackTrace()	Returns an array containing the stack trace for the invoking thread.
Thread.State getState()	Returns the invoking thread's state.
final ThreadGroup getThreadGroup()	Returns the ThreadGroup object of which the invoking thread is a member.
Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()	Returns the invoking thread's uncaught exception handler.
static boolean holdsLock(Object <i>ob</i>)	Returns true if the invoking thread owns the lock on <i>ob</i> . Returns false otherwise.
void interrupt()	Interrupts the thread.
static boolean interrupted()	Returns true if the currently executing thread has been interrupted. Otherwise, it returns false .
final boolean isAlive()	Returns true if the thread is still active. Otherwise, it returns false .
final boolean isDaemon()	Returns true if the thread is a daemon thread. Otherwise, it returns false .
boolean isInterrupted()	Returns true if the invoking thread has been interrupted. Otherwise, it returns false .
final void join() throws InterruptedException	Waits until the thread terminates.
final void join(long <i>milliseconds</i>) throws InterruptedException	Waits up to the specified number of milliseconds for the thread on which it is called to terminate.
final void join(long <i>milliseconds</i> , int <i>nanoseconds</i>) throws InterruptedException	Waits up to the specified number of milliseconds plus nanoseconds for the thread on which it is called to terminate.
void run()	Begins execution of a thread.
void setContextClassLoader(ClassLoader <i>cl</i>)	Sets the context class loader that will be used by the invoking thread to <i>cl</i> .
final void setDaemon(boolean <i>state</i>)	Flags the thread as a daemon thread.
static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler <i>e</i>)	Sets the default uncaught exception handler to <i>e</i> .

Table 17-18 The Methods Defined by **Thread** (*continued*)

Method	Description
final void setName(String <i>threadName</i>)	Sets the name of the thread to that specified by <i>threadName</i> .
final void setPriority(int <i>priority</i>)	Sets the priority of the thread to that specified by <i>priority</i> .
void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler <i>e</i>)	Sets the invoking thread's default uncaught exception handler to <i>e</i> .
static void sleep(long <i>milliseconds</i>) throws InterruptedException	Suspends execution of the thread for the specified number of milliseconds.
static void sleep(long <i>milliseconds</i> , int <i>nanoseconds</i>) throws InterruptedException	Suspends execution of the thread for the specified number of milliseconds plus nanoseconds.
void start()	Starts execution of the thread.
String toString()	Returns the string equivalent of a thread.
static void yield()	The calling thread offers to yield the CPU to another thread.

Table 17-18 The Methods Defined by **Thread** (*continued*)

ThreadGroup

ThreadGroup creates a group of threads. It defines these two constructors:

```
ThreadGroup(String groupName)
ThreadGroup(ThreadGroup parentOb, String groupName)
```

For both forms, *groupName* specifies the name of the thread group. The first version creates a new group that has the current thread as its parent. In the second form, the parent is specified by *parentOb*. The non-deprecated methods defined by **ThreadGroup** are shown in Table 17-19.

Method	Description
int activeCount()	Returns the approximate number of active threads in the invoking group (including those in subgroups).
int activeGroupCount()	Returns the approximate number of active groups (including subgroups) for which the invoking thread is a parent.
final void checkAccess()	Causes the security manager to verify that the invoking thread may access and/or change the group on which <code>checkAccess()</code> is called.
final void destroy()	Destroys the thread group (and any child groups) on which it is called.

Table 17-19 The Methods Defined by **ThreadGroup**

Method	Description
int enumerate(Thread <i>group</i> [])	Puts the active threads that comprise the invoking thread group (including those in subgroups) into the <i>group</i> array.
int enumerate(Thread <i>group</i> [], boolean <i>all</i>)	Puts the active threads that comprise the invoking thread group into the <i>group</i> array. If <i>all</i> is true , then threads in all subgroups of the thread are also put into <i>group</i> .
int enumerate(ThreadGroup <i>group</i> [])	Puts the active subgroups (including subgroups of subgroups and so on) of the invoking thread group into the <i>group</i> array.
int enumerate(ThreadGroup <i>group</i> [], boolean <i>all</i>)	Puts the active subgroups of the invoking thread group into the <i>group</i> array. If <i>all</i> is true , then all active subgroups of the subgroups (and so on) are also put into <i>group</i> .
final int getMaxPriority()	Returns the maximum priority setting for the group.
final String getName()	Returns the name of the group.
final ThreadGroup getParent()	Returns null if the invoking ThreadGroup object has no parent. Otherwise, it returns the parent of the invoking object.
final void interrupt()	Invokes the interrupt() method of all threads in the group and any subgroups.
final boolean isDaemon()	Returns true if the group is a daemon group. Otherwise, it returns false .
boolean isDestroyed()	Returns true if the group has been destroyed. Otherwise, it returns false .
void list()	Displays information about the group.
final boolean parentOf(ThreadGroup <i>group</i>)	Returns true if the invoking thread is the parent of <i>group</i> (or <i>group</i> , itself). Otherwise, it returns false .
final void setDaemon(boolean <i>isDaemon</i>)	If <i>isDaemon</i> is true , then the invoking group is flagged as a daemon group.
final void setMaxPriority(int <i>priority</i>)	Sets the maximum priority of the invoking group to <i>priority</i> .
String toString()	Returns the string equivalent of the group.
void uncaughtException(Thread <i>thread</i> , Throwable <i>e</i>)	This method is called when an exception goes uncaught.

Table 17-19 The Methods Defined by **ThreadGroup** (*continued*)

Thread groups offer a convenient way to manage groups of threads as a unit. This is particularly valuable in situations in which you want to suspend and resume a number of related threads. For example, imagine a program in which one set of threads is used for printing a document, another set is used to display the document on the screen, and another set saves the document to a disk file. If printing is aborted, you will want an easy way to stop all threads related to printing. Thread groups offer this convenience. The following program, which creates two thread groups of two threads each, illustrates this usage:

```
// Demonstrate thread groups.
class NewThread extends Thread {
    boolean suspendFlag;

    NewThread(String threadname, ThreadGroup tgOb) {
        super(tgOb, threadname);
        System.out.println("New thread: " + this);
        suspendFlag = false;
        start(); // Start the thread
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(getName() + ": " + i);
                Thread.sleep(1000);
                synchronized(this) {
                    while(suspendFlag) {
                        wait();
                    }
                }
            }
        } catch (Exception e) {
            System.out.println("Exception in " + getName());
        }
        System.out.println(getName() + " exiting.");
    }

    synchronized void mysuspend() {
        suspendFlag = true;
    }

    synchronized void myresume() {
        suspendFlag = false;
        notify();
    }
}

class ThreadGroupDemo {
    public static void main(String args[]) {
        ThreadGroup groupA = new ThreadGroup("Group A");
        ThreadGroup groupB = new ThreadGroup("Group B");
```

```
NewThread ob1 = new NewThread("One", groupA);
NewThread ob2 = new NewThread("Two", groupA);
NewThread ob3 = new NewThread("Three", groupB);
NewThread ob4 = new NewThread("Four", groupB);

System.out.println("\nHere is output from list():");
groupA.list();
groupB.list();
System.out.println();

System.out.println("Suspending Group A");
Thread tga[] = new Thread[groupA.activeCount()];
groupA.enumerate(tga); // get threads in group
for(int i = 0; i < tga.length; i++) {
    ((NewThread)tga[i]).mysuspend(); // suspend each thread
}

try {
    Thread.sleep(4000);
} catch (InterruptedException e) {
    System.out.println("Main thread interrupted.");
}

System.out.println("Resuming Group A");
for(int i = 0; i < tga.length; i++) {
    ((NewThread)tga[i]).myresume(); // resume threads in group
}

// wait for threads to finish
try {
    System.out.println("Waiting for threads to finish.");
    ob1.join();
    ob2.join();
    ob3.join();
    ob4.join();
} catch (Exception e) {
    System.out.println("Exception in Main thread");
}

System.out.println("Main thread exiting.");
}
```

Sample output from this program is shown here (the precise output you see may differ):

```
New thread: Thread[One,5,Group A]
New thread: Thread[Two,5,Group A]
New thread: Thread[Three,5,Group B]
New thread: Thread[Four,5,Group B]
Here is output from list():
java.lang.ThreadGroup[name=Group A,maxpri=10]
    Thread[One,5,Group A]
    Thread[Two,5,Group A]
```

```
java.lang.ThreadGroup [name=Group B,maxpri=10]
    Thread [Three,5,Group B]
        Thread [Four,5,Group B]
Suspending Group A
Three: 5
Four: 5
Three: 4
Four: 4
Three: 3
Four: 3
Three: 2
Four: 2
Resuming Group A
Waiting for threads to finish.
One: 5
Two: 5
Three: 1
Four: 1
One: 4
Two: 4
Three exiting.
Four exiting.
One: 3
Two: 3
One: 2
Two: 2
One: 1
Two: 1
One exiting.
Two exiting.
Main thread exiting.
```

Inside the program, notice that thread group A is suspended for four seconds. As the output confirms, this causes threads One and Two to pause, but threads Three and Four continue running. After the four seconds, threads One and Two are resumed. Notice how thread group A is suspended and resumed. First, the threads in group A are obtained by calling `enumerate()` on group A. Then, each thread is suspended by iterating through the resulting array. To resume the threads in A, the list is again traversed and each thread is resumed. One last point: This example uses the recommended approach to suspending and resuming threads. It does not rely upon the deprecated methods `suspend()` and `resume()`.

ThreadLocal and InheritableThreadLocal

Java defines two additional thread-related classes in `java.lang`:

- **ThreadLocal** Used to create thread local variables. Each thread will have its own copy of a thread local variable.
- **InheritableThreadLocal** Creates thread local variables that may be inherited.

Package

Package encapsulates version data associated with a package. **Package** version information is becoming more important because of the proliferation of packages and because a Java program may need to know what version of a package is available. The methods defined by **Package** are shown in Table 17-20. The following program demonstrates **Package**, displaying the packages about which the program currently is aware:

```
// Demonstrate Package
class PkgTest {
    public static void main(String args[]) {
        Package pkgs[];

        pkgs = Package.getPackages();

        for(int i=0; i < pkgs.length; i++)
            System.out.println(
                pkgs[i].getName() + " " +
                pkgs[i].getImplementationTitle() + " " +
                pkgs[i].getImplementationVendor() + " " +
                pkgs[i].getImplementationVersion()
            );
    }
}
```

Method	Description
<A extends Annotation> A getAnnotation(Class<A> annoType)	Returns an Annotation object that contains the annotation associated with <i>annoType</i> for the invoking object.
Annotation[] getAnnotations()	Returns all annotations associated with the invoking object in an array of Annotation objects. Returns a reference to this array.
<A extends Annotation> A[] getAnnotationsByType(Class<A> annoType)	Returns an array of the annotations (including repeated annotations) of <i>annoType</i> associated with the invoking object. (Added by JDK 8.)
<A extends Annotation> A getDeclaredAnnotation(Class<A> annoType)	Returns an Annotation object that contains the non-inherited annotation associated with <i>annoType</i> . (Added by JDK 8.)
Annotation[] getDeclaredAnnotations()	Returns an Annotation object for all the annotations that are declared by the invoking object. (Inherited annotations are ignored.)
<A extends Annotation> A[] getDeclaredAnnotationsByType(Class<A> annoType)	Returns an array of the non-inherited annotations (including repeated annotations) of <i>annoType</i> associated with the invoking object. (Added by JDK 8.)
String getImplementationTitle()	Returns the title of the invoking package.

Table 17-20 The Methods Defined by **Package**

Method	Description
String getImplementationVendor()	Returns the name of the implementor of the invoking package.
String getImplementationVersion()	Returns the version number of the invoking package.
String getName()	Returns the name of the invoking package.
static Package getPackage(String <i>pkgName</i>)	Returns a Package object with the name specified by <i>pkgName</i> .
static Package[] getPackages()	Returns all packages about which the invoking program is currently aware.
String getSpecificationTitle()	Returns the title of the invoking package's specification.
String getSpecificationVendor()	Returns the name of the owner of the specification for the invoking package.
String getSpecificationVersion()	Returns the invoking package's specification version number.
int hashCode()	Returns the hash code for the invoking package.
boolean isAnnotationPresent(Class<? extends Annotation> <i>anno</i>)	Returns true if the annotation described by <i>anno</i> is associated with the invoking object. Returns false otherwise.
boolean isCompatibleWith(String <i>verNum</i>) throws NumberFormatException	Returns true if <i>verNum</i> is less than or equal to the invoking package's version number.
boolean isSealed()	Returns true if the invoking package is sealed. Returns false otherwise.
boolean isSealed(URL <i>url</i>)	Returns true if the invoking package is sealed relative to <i>url</i> . Returns false otherwise.
String toString()	Returns the string equivalent of the invoking package.

Table 17-20 The Methods Defined by **Package** (*continued*)

RuntimePermission

The **RuntimePermission** relates to Java's security mechanism and is not examined further here.

Throwable

The **Throwable** class supports Java's exception-handling system and is the class from which all exception classes are derived. It is discussed in Chapter 10.

SecurityManager

The **SecurityManager** supports Java's security system. A reference to the current security manager can be obtained by calling **getSecurityManager()** defined by the **System** class.

StackTraceElement

The **StackTraceElement** class describes a single *stack frame*, which is an individual element of a stack trace when an exception occurs. Each stack frame represents an *execution point*, which includes such things as the name of the class, the name of the method, the name of the file, and the source-code line number. An array of **StackTraceElements** is returned by the `getStackTrace()` method of the **Throwable** class.

StackTraceElement has one constructor:

```
StackTraceElement(String className, String methName, string fileName, int line)
```

Here, the name of the class is specified by *className*, the name of the method is specified in *methName*, the name of the file is specified by *fileName*, and the line number is passed in *line*. If there is no valid line number, use a negative value for *line*. Furthermore, a value of -2 for *line* indicates that this frame refers to a native method.

The methods supported by **StackTraceElement** are shown in Table 17-21. These methods give you programmatic access to a stack trace.

Method	Description
<code>boolean equals(Object ob)</code>	Returns true if the invoking StackTraceElement is the same as the one passed in <i>ob</i> . Otherwise, it returns false .
<code>String getClassName()</code>	Returns the name of the class in which the execution point described by the invoking StackTraceElement occurred.
<code>String getFileName()</code>	Returns the name of the file in which the source code of the execution point described by the invoking StackTraceElement is stored.
<code>int getLineNumber()</code>	Returns the source-code line number at which the execution point described by the invoking StackTraceElement occurred. In some situations, the line number will not be available, in which case a negative value is returned.
<code>String getMethodName()</code>	Returns the name of the method in which the execution point described by the invoking StackTraceElement occurred.
<code>int hashCode()</code>	Returns the hash code for the invoking StackTraceElement .
<code>boolean isNativeMethod()</code>	Returns true if the execution point described by the invoking StackTraceElement occurred in a native method. Otherwise, it returns false .
<code>String toString()</code>	Returns the String equivalent of the invoking sequence.

Table 17-21 The Methods Defined by **StackTraceElement**

Enum

As described in Chapter 12, an enumeration is a list of named constants. (Recall that an enumeration is created by using the keyword **enum**.) All enumerations automatically inherit **Enum**. **Enum** is a generic class that is declared as shown here:

```
class Enum<E extends Enum<E>>
```

Here, **E** stands for the enumeration type. **Enum** has no public constructors.

Enum defines several methods that are available for use by all enumerations, which are shown in Table 17-22.

Method	Description
protected final Object clone() throws CloneNotSupportedException	Invoking this method causes a CloneNotSupportedException to be thrown. This prevents enumerations from being cloned.
final int compareTo(E e)	Compares the ordinal value of two constants of the same enumeration. Returns a negative value if the invoking constant has an ordinal value less than e's, zero if the two ordinal values are the same, and a positive value if the invoking constant has an ordinal value greater than e's.
final boolean equals(Object obj)	Returns true if obj and the invoking object refer to the same constant.
final Class<E> getDeclaringClass()	Returns the type of enumeration of which the invoking constant is a member.
final int hashCode()	Returns the hash code for the invoking object.
final String name()	Returns the unaltered name of the invoking constant.
final int ordinal()	Returns a value that indicates an enumeration constant's position in the list of constants.
String toString()	Returns the name of the invoking constant. This name may differ from the one used in the enumeration's declaration.
static <T extends Enum<T>> T valueOf(Class<T> e-type, String name)	Returns the constant associated with name in the enumeration type specified by e-type.

Table 17-22 The Methods Defined by **Enum**

ClassValue

ClassValue can be used to associate a value with a type. It is a generic type defined like this:

```
Class ClassValue<T>
```

It is designed for highly specialized uses, not for normal programming.

The CharSequence Interface

The **CharSequence** interface defines methods that grant read-only access to a sequence of characters. These methods are shown in Table 17-23. This interface is implemented by **String**, **StringBuffer**, and **StringBuilder**, among others.

The Comparable Interface

Objects of classes that implement **Comparable** can be ordered. In other words, classes that implement **Comparable** contain objects that can be compared in some meaningful manner. **Comparable** is generic and is declared like this:

```
interface Comparable<T>
```

Here, **T** represents the type of objects being compared.

Method	Description
char charAt(int <i>idx</i>)	Returns the character at the index specified by <i>idx</i> .
default IntStream chars()	Returns a stream (in the form of an IntStream) to the characters in the invoking object. (Added by JDK 8.)
default IntStream codePoints()	Returns a stream (in the form of an IntStream) to the code points in the invoking object. (Added by JDK 8.)
int length()	Returns the number of characters in the invoking sequence.
CharSequence subSequence(int <i>startIdx</i> , int <i>stopIdx</i>)	Returns a subset of the invoking sequence beginning at <i>startIdx</i> and ending at <i>stopIdx</i> -1.
String toString()	Returns the String equivalent of the invoking sequence.

Table 17-23 The Methods Defined by **CharSequence**

The **Comparable** interface declares one method that is used to determine what Java calls the *natural ordering* of instances of a class. The signature of the method is shown here:

```
int compareTo(T obj)
```

This method compares the invoking object with *obj*. It returns 0 if the values are equal. A negative value is returned if the invoking object has a lower value. Otherwise, a positive value is returned.

This interface is implemented by several of the classes already reviewed in this book. Specifically, the **Byte**, **Character**, **Double**, **Float**, **Long**, **Short**, **String**, and **Integer** classes define a **compareTo()** method. So does **Enum**.

The Appendable Interface

Objects of a class that implements **Appendable** can have a character or character sequences appended to it. **Appendable** defines these three methods:

```
Appendable append(char ch) throws IOException
```

```
Appendable append(CharSequence chars) throws IOException
```

```
Appendable append(CharSequence chars, int begin, int end) throws IOException
```

In the first form, the character *ch* is appended to the invoking object. In the second form, the character sequence *chars* is appended to the invoking object. The third form allows you to indicate a portion (the characters running from *begin* through *end*-1) of the sequence specified by *chars*. In all cases, a reference to the invoking object is returned.

The Iterable Interface

Iterable must be implemented by any class whose objects will be used by the for-each version of the **for** loop. In other words, in order for an object to be used within a for-each style **for** loop, its class must implement **Iterable**. **Iterable** is a generic interface that has this declaration:

```
interface Iterable<T>
```

Here, **T** is the type of the object being iterated. It defines one abstract method, **iterator()**, which is shown here:

```
Iterator<T> iterator()
```

It returns an iterator to the elements contained in the invoking object.

Beginning with JDK 8, **Iterable** also defines two default methods. The first is called **forEach()**:

```
default void forEach(Consumer<? super T> action)
```

For each element being iterated, **forEach()** executes the code specified by *action*. (**Consumer** is a functional interface added by JDK 8 and defined in **java.util.function**. See Chapter 19.)

The second default method is **spliterator()**, shown next:

```
default Spliterator<T> spliterator()
```

It returns a **Spliterator** to the sequence being iterated. (See Chapters 18 and 29 for details on spliterators.)

NOTE Iterators are described in detail in Chapter 18.

The Readable Interface

The **Readable** interface indicates that an object can be used as a source for characters. It defines one method called **read()**, which is shown here:

```
int read(CharBuffer buf) throws IOException
```

This method reads characters into *buf*. It returns the number of characters read, or *-1* if an EOF is encountered.

The AutoCloseable Interface

AutoCloseable provides support for the **try-with-resources** statement, which implements what is sometimes referred to as *automatic resource management* (ARM). The **try-with-resources** statement automates the process of releasing a resource (such as a stream) when it is no longer needed. (See Chapter 13 for details.) Only objects of classes that implement **AutoCloseable** can be used with **try-with-resources**. The **AutoCloseable** interface defines only the **close()** method, which is shown here:

```
void close() throws Exception
```

This method closes the invoking object, releasing any resources that it may hold. It is automatically called at the end of a **try-with-resources** statement, thus eliminating the need to explicitly invoke **close()**. **AutoCloseable** is implemented by several classes, including all of the I/O classes that open a stream that can be closed.

The Thread.UncaughtExceptionHandler Interface

The static **Thread.UncaughtExceptionHandler** interface is implemented by classes that want to handle uncaught exceptions. It is implemented by **ThreadGroup**. It declares only one method, which is shown here:

```
void uncaughtException(Thread thrd, Throwable exc)
```

Here, *thrd* is a reference to the thread that generated the exception and *exc* is a reference to the exception.

The java.lang Subpackages

Java defines several subpackages of **java.lang**:

- `java.lang.annotation`
- `java.lang.instrument`
- `java.lang.invoke`

- `java.lang.management`
- `java.lang.ref`
- `java.lang.reflect`

Each is briefly described here.

java.lang.annotation

Java's annotation facility is supported by `java.lang.annotation`. It defines the `Annotation` interface, the `ElementType` and `RetentionPolicy` enumerations, and several predefined annotations. Annotations are described in Chapter 12.

java.lang.instrument

`java.lang.instrument` defines features that can be used to add instrumentation to various aspects of program execution. It defines the `Instrumentation` and `ClassFileTransformer` interfaces, and the `ClassDefinition` class.

java.lang.invoke

`java.lang.invoke` supports dynamic languages. It includes classes such as `CallSite`, `MethodHandle`, and `MethodType`.

java.lang.management

The `java.lang.management` package provides management support for the JVM and the execution environment. Using the features in `java.lang.management`, you can observe and manage various aspects of program execution.

java.lang.ref

You learned earlier that the garbage collection facilities in Java automatically determine when no references exist to an object. The object is then assumed to be no longer needed and its memory is reclaimed. The classes in the `java.lang.ref` package provide more flexible control over the garbage collection process.

java.lang.reflect

Reflection is the ability of a program to analyze code at run time. The `java.lang.reflect` package provides the ability to obtain information about the fields, constructors, methods, and modifiers of a class. Among other reasons, you need this information to build software tools that enable you to work with Java Beans components. The tools use reflection to determine dynamically the characteristics of a component. Reflection was introduced in Chapter 12 and is also examined in Chapter 30.

`java.lang.reflect` defines several classes, including `Method`, `Field`, and `Constructor`. It also defines several interfaces, including `AnnotatedElement`, `Member`, and `Type`. In addition, the `java.lang.reflect` package includes the `Array` class that enables you to create and access arrays dynamically.

CHAPTER

18

java.util Part 1: The Collections Framework

This chapter begins our examination of **java.util**. This important package contains a large assortment of classes and interfaces that support a broad range of functionality. For example, **java.util** has classes that generate pseudorandom numbers, manage date and time, observe events, manipulate sets of bits, tokenize strings, and handle formatted data. The **java.util** package also contains one of Java's most powerful subsystems: the *Collections Framework*. The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects. It merits close attention by all programmers.

Because **java.util** contains a wide array of functionality, it is quite large. Here is a list of its top-level classes:

AbstractCollection	FormattableFlags	Properties
AbstractList	Formatter	PropertyPermission
AbstractMap	GregorianCalendar	PropertyResourceBundle
AbstractQueue	HashMap	Random
AbstractSequentialList	HashSet	ResourceBundle
AbstractSet	Hashtable	Scanner
ArrayDeque	IdentityHashMap	ServiceLoader
ArrayList	IntSummaryStatistics (Added by JDK 8.)	SimpleTimeZone
Arrays	LinkedHashMap	Spliterators (Added by JDK 8.)
Base64 (Added by JDK 8.)	LinkedHashSet	SplittableRandom (Added by JDK 8.)
BitSet	LinkedList	Stack
Calendar	ListResourceBundle	StringJoiner (Added by JDK 8.)

Collections	Locale	StringTokenizer
Currency	LongSummaryStatistics (Added by JDK 8.)	Timer
Date	Objects	TimerTask
Dictionary	Observable	TimeZone
DoubleSummaryStatistics (Added by JDK 8.)	Optional (Added by JDK 8.)	TreeMap
EnumMap	OptionalDouble (Added by JDK 8.)	TreeSet
EnumSet	OptionalInt (Added by JDK 8.)	UUID
EventListenerProxy	OptionalLong (Added by JDK 8.)	Vector
EventObject	PriorityQueue	WeakHashMap

The interfaces defined by **java.util** are shown next:

Collection	Map.Entry	Set
Comparator	NavigableMap	SortedMap
Deque	NavigableSet	SortedSet
Enumeration	Observer	Spliterator (Added by JDK 8.)
EventListener	PrimitiveIterator (Added by JDK 8.)	Spliterator.OfDouble (Added by JDK 8.)
Formattable	PrimitiveIterator.OfDouble (Added by JDK 8.)	Spliterator.OfInt (Added by JDK 8.)
Iterator	PrimitiveIterator.OfInt (Added by JDK 8.)	Spliterator.OfLong (Added by JDK 8.)
List	PrimitiveIterator.OfLong (Added by JDK 8.)	Spliterator.OfPrimitive (Added by JDK 8.)
ListIterator	Queue	
Map	RandomAccess	

Because of its size, the description of **java.util** is broken into two chapters. This chapter examines those members of **java.util** that are part of the Collections Framework. Chapter 18 discusses its other classes and interfaces.

Collections Overview

The Java Collections Framework standardizes the way in which groups of objects are handled by your programs. Collections were not part of the original Java release, but were added by J2SE 1.2. Prior to the Collections Framework, Java provided ad hoc classes such as **Dictionary**, **Vector**, **Stack**, and **Properties** to store and manipulate groups of objects. Although these

classes were quite useful, they lacked a central, unifying theme. The way that you used **Vector** was different from the way that you used **Properties**, for example. Also, this early, ad hoc approach was not designed to be easily extended or adapted. Collections are an answer to these (and other) problems.

The Collections Framework was designed to meet several goals. First, the framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient. You seldom, if ever, need to code one of these “data engines” manually. Second, the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability. Third, extending and/or adapting a collection had to be easy. Toward this end, the entire Collections Framework is built upon a set of standard interfaces. Several standard implementations (such as **LinkedList**, **HashSet**, and **TreeSet**) of these interfaces are provided that you may use as-is. You may also implement your own collection, if you choose. Various special-purpose implementations are created for your convenience, and some partial implementations are provided that make creating your own collection class easier. Finally, mechanisms were added that allow the integration of standard arrays into the Collections Framework.

Algorithms are another important part of the collection mechanism. Algorithms operate on collections and are defined as static methods within the **Collections** class. Thus, they are available for all collections. Each collection class need not implement its own versions. The algorithms provide a standard means of manipulating collections.

Another item closely associated with the Collections Framework is the **Iterator** interface. An *iterator* offers a general-purpose, standardized way of accessing the elements within a collection, one at a time. Thus, an iterator provides a means of *enumerating the contents of a collection*. Because each collection provides an iterator, the elements of any collection class can be accessed through the methods defined by **Iterator**. Thus, with only small changes, the code that cycles through a set can also be used to cycle through a list, for example.

JDK 8 adds another type of iterator called a *spliterator*. In brief, spliterators are iterators that provide support for parallel iteration. The interfaces that support spliterators are **Spliterator** and several nested interfaces that support primitive types. JDK 8 also adds iterator interfaces designed for use with primitive types, such as **PrimitiveIterator** and **PrimitiveIterator.OfDouble**.

In addition to collections, the framework defines several map interfaces and classes. *Maps* store key/value pairs. Although maps are part of the Collections Framework, they are not “collections” in the strict use of the term. You can, however, obtain a *collection-view* of a map. Such a view contains the elements from the map stored in a collection. Thus, you can process the contents of a map as a collection, if you choose.

The collection mechanism was retrofitted to some of the original classes defined by **java.util** so that they too could be integrated into the new system. It is important to understand that although the addition of collections altered the architecture of many of the original utility classes, it did not cause the deprecation of any. Collections simply provide a better way of doing several things.

NOTE If you are familiar with C++, then you will find it helpful to know that the Java collections technology is similar in spirit to the Standard Template Library (STL) defined by C++. What C++ calls a container, Java calls a collection. However, there are significant differences between the Collections Framework and the STL. It is important to not jump to conclusions.

JDK 5 Changed the Collections Framework

When JDK 5 was released, some fundamental changes were made to the Collections Framework that significantly increased its power and streamlined its use. These changes include the addition of generics, autoboxing/unboxing, and the for-each style **for** loop. Although JDK 8 is three major Java releases after JDK 5, the effects of the JDK 5 features were so profound that they still warrant special attention. The main reason is that you may encounter pre-JDK 5 code. Understanding the effects and reasons for the changes is important if you will be maintaining or updating older code.

Generics Fundamentally Changed the Collections Framework

The addition of generics caused a significant change to the Collections Framework because the entire Collections Framework was reengineered for it. All collections are now generic, and many of the methods that operate on collections take generic type parameters. Simply put, the addition of generics affected every part of the Collections Framework.

Generics added the one feature that collections had been missing: type safety. Prior to generics, all collections stored **Object** references, which meant that any collection could store any type of object. Thus, it was possible to accidentally store incompatible types in a collection. Doing so could result in run-time type mismatch errors. With generics, it is possible to explicitly state the type of data being stored, and run-time type mismatch errors can be avoided.

Although the addition of generics changed the declarations of most of its classes and interfaces, and several of their methods, overall, the Collections Framework still works the same as it did prior to generics. Of course, to gain the advantages that generics bring collections, older code will need to be rewritten. This is also important because pre-generics code will generate warning messages when compiled by a modern Java compiler. To eliminate these warnings, you will need to add type information to all your collections code.

Autoboxing Facilitates the Use of Primitive Types

Autoboxing/unboxing facilitates the storing of primitive types in collections. As you will see, a collection can store only references, not primitive values. In the past, if you wanted to store a primitive value, such as an **int**, in a collection, you had to manually box it into its type wrapper. When the value was retrieved, it needed to be manually unboxed (by using an explicit cast) into its proper primitive type. Because of autoboxing/unboxing, Java can automatically perform the proper boxing and unboxing needed when storing or retrieving primitive types. There is no need to manually perform these operations.

The For-Each Style for Loop

All collection classes in the Collections Framework were retrofitted to implement the **Iterable** interface, which means that a collection can be cycled through by use of the for-each style **for** loop. In the past, cycling through a collection required the use of an iterator (described later in this chapter), with the programmer manually constructing the loop. Although iterators are still needed for some uses, in many cases, iterator-based loops can be replaced by **for** loops.

The Collection Interfaces

The Collections Framework defines several core interfaces. This section provides an overview of each interface. Beginning with the collection interfaces is necessary because they determine the fundamental nature of the collection classes. Put differently, the concrete classes simply provide different implementations of the standard interfaces. The interfaces that underpin collections are summarized in the following table:

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
Deque	Extends Queue to handle a double-ended queue.
List	Extends Collection to handle sequences (lists of objects).
NavigableSet	Extends SortedSet to handle retrieval of elements based on closest-match searches.
Queue	Extends Collection to handle special types of lists in which elements are removed only from the head.
Set	Extends Collection to handle sets, which must contain unique elements.
SortedSet	Extends Set to handle sorted sets.

In addition to the collection interfaces, collections also use the **Comparator**, **RandomAccess**, **Iterator**, and **ListIterator** interfaces, which are described in depth later in this chapter. Beginning with JDK 8, **Spliterator** can also be used. Briefly, **Comparator** defines how two objects are compared; **Iterator**, **ListIterator**, and **Spliterator** enumerate the objects within a collection. By implementing **RandomAccess**, a list indicates that it supports efficient, random access to its elements.

To provide the greatest flexibility in their use, the collection interfaces allow some methods to be optional. The optional methods enable you to modify the contents of a collection. Collections that support these methods are called *modifiable*. Collections that do not allow their contents to be changed are called *unmodifiable*. If an attempt is made to use one of these methods on an unmodifiable collection, an **UnsupportedOperationException** is thrown. All the built-in collections are modifiable.

The following sections examine the collection interfaces.

The Collection Interface

The **Collection** interface is the foundation upon which the Collections Framework is built because it must be implemented by any class that defines a collection. **Collection** is a generic interface that has this declaration:

```
interface Collection<E>
```

Here, **E** specifies the type of objects that the collection will hold. **Collection** extends the **Iterable** interface. This means that all collections can be cycled through by use of the for-each style **for** loop. (Recall that only classes that implement **Iterable** can be cycled through by the **for**.)

Collection declares the core methods that all collections will have. These methods are summarized in Table 18-1. Because all collections implement **Collection**, familiarity with its methods is necessary for a clear understanding of the framework. Several of these methods can throw an **UnsupportedOperationException**. As explained, this occurs if a collection cannot be modified. A **ClassCastException** is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a collection. A **NullPointerException** is thrown if an attempt is made to store a **null** object and **null** elements are not allowed in the collection. An **IllegalArgumentException** is thrown if an invalid argument is used. An **IllegalStateException** is thrown if an attempt is made to add an element to a fixed-length collection that is full.

Method	Description
boolean add(E <i>obj</i>)	Adds <i>obj</i> to the invoking collection. Returns true if <i>obj</i> was added to the collection. Returns false if <i>obj</i> is already a member of the collection and the collection does not allow duplicates.
boolean addAll(Collection<? extends E> <i>c</i>)	Adds all the elements of <i>c</i> to the invoking collection. Returns true if the collection changed (i.e., the elements were added). Otherwise, returns false .
void clear()	Removes all elements from the invoking collection.
boolean contains(Object <i>obj</i>)	Returns true if <i>obj</i> is an element of the invoking collection. Otherwise, returns false .
boolean containsAll(Collection<?> <i>c</i>)	Returns true if the invoking collection contains all elements of <i>c</i> . Otherwise, returns false .
boolean equals(Object <i>obj</i>)	Returns true if the invoking collection and <i>obj</i> are equal. Otherwise, returns false .
int hashCode()	Returns the hash code for the invoking collection.
boolean isEmpty()	Returns true if the invoking collection is empty. Otherwise, returns false .
Iterator<E> iterator()	Returns an iterator for the invoking collection.
default Stream<E> parallelStream()	Returns a stream that uses the invoking collection as its source for elements. If possible, the stream supports parallel operations. (Added by JDK 8.)
boolean remove(Object <i>obj</i>)	Removes one instance of <i>obj</i> from the invoking collection. Returns true if the element was removed. Otherwise, returns false .
boolean removeAll(Collection<?> <i>c</i>)	Removes all elements of <i>c</i> from the invoking collection. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false .
default boolean removeIf(Predicate<? super E> <i>predicate</i>)	Removes from the invoking collection those elements that satisfy the condition specified by <i>predicate</i> . (Added by JDK 8.)

Table 18-1 The Methods Declared by **Collection**

Method	Description
boolean retainAll(Collection<?> <i>c</i>)	Removes all elements from the invoking collection except those in <i>c</i> . Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false .
int size()	Returns the number of elements held in the invoking collection.
default Spliterator<E> spliterator()	Returns a spliterator to the invoking collections. (Added by JDK 8.)
default Stream<E> stream()	Returns a stream that uses the invoking collection as its source for elements. The stream is sequential. (Added by JDK 8.)
Object[] toArray()	Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
<T> T[] toArray(T <i>array</i> [])	Returns an array that contains the elements of the invoking collection. The array elements are copies of the collection elements. If the size of <i>array</i> equals the number of elements, these are returned in <i>array</i> . If the size of <i>array</i> is less than the number of elements, a new array of the necessary size is allocated and returned. If the size of <i>array</i> is greater than the number of elements, the array element following the last collection element is set to null . An ArrayStoreException is thrown if any collection element has a type that is not a subtype of <i>array</i> .

Table 18-1 The Methods Declared by **Collection** (continued)

Objects are added to a collection by calling **add()**. Notice that **add()** takes an argument of type **E**, which means that objects added to a collection must be compatible with the type of data expected by the collection. You can add the entire contents of one collection to another by calling **addAll()**.

You can remove an object by using **remove()**. To remove a group of objects, call **removeAll()**. You can remove all elements except those of a specified group by calling **retainAll()**. Beginning with JDK 8, to remove an element only if it satisfies some condition, you can use **removeIf()**. (**Predicate** is a functional interface added by JDK 8. See Chapter 19.) To empty a collection, call **clear()**.

You can determine whether a collection contains a specific object by calling **contains()**. To determine whether one collection contains all the members of another, call **containsAll()**. You can determine when a collection is empty by calling **isEmpty()**. The number of elements currently held in a collection can be determined by calling **size()**.

The **toArray()** methods return an array that contains the elements stored in the invoking collection. The first returns an array of **Object**. The second returns an array of elements that have the same type as the array specified as a parameter. Normally, the second form is more convenient because it returns the desired array type. These methods are more important than it might at first seem. Often, processing the contents of a

collection by using array-like syntax is advantageous. By providing a pathway between collections and arrays, you can have the best of both worlds.

Two collections can be compared for equality by calling `equals()`. The precise meaning of “equality” may differ from collection to collection. For example, you can implement `equals()` so that it compares the values of elements stored in the collection. Alternatively, `equals()` can compare references to those elements.

Another important method is `iterator()`, which returns an iterator to a collection. The new `spliterator()` method returns a spliterator to the collection. Iterators are frequently used when working with collections. Finally, the `stream()` and `parallelStream()` methods return a `Stream` that uses the collection as a source of elements. (See Chapter 29 for a detailed discussion of the new `Stream` interface.)

The List Interface

The `List` interface extends `Collection` and declares the behavior of a collection that stores a sequence of elements. Elements can be inserted or accessed by their position in the list, using a zero-based index. A list may contain duplicate elements. `List` is a generic interface that has this declaration:

```
interface List<E>
```

Here, `E` specifies the type of objects that the list will hold.

In addition to the methods defined by `Collection`, `List` defines some of its own, which are summarized in Table 18-2. Note again that several of these methods will throw an `UnsupportedOperationException` if the list cannot be modified, and a `ClassCastException` is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a list. Also, several methods will throw an `IndexOutOfBoundsException` if an invalid index is used. A `NullPointerException` is thrown if an attempt is made to store a `null` object and `null` elements are not allowed in the list. An `IllegalArgumentException` is thrown if an invalid argument is used.

To the versions of `add()` and `addAll()` defined by `Collection`, `List` adds the methods `add(int, E)` and `addAll(int, Collection)`. These methods insert elements at the specified index. Also, the semantics of `add(E)` and `addAll(Collection)` defined by `Collection` are changed by `List` so that they add elements to the end of the list. You can modify each element in the collection by using `replaceAll()`. (`UnaryOperator` is a functional interface added by JDK 8. See Chapter 19.)

To obtain the object stored at a specific location, call `get()` with the index of the object. To assign a value to an element in the list, call `set()`, specifying the index of the object to be changed. To find the index of an object, use `indexOf()` or `lastIndexOf()`.

You can obtain a sublist of a list by calling `subList()`, specifying the beginning and ending indexes of the sublist. As you can imagine, `subList()` makes list processing quite convenient. One way to sort a list is with the `sort()` method defined by `List`.

The Set Interface

The `Set` interface defines a set. It extends `Collection` and specifies the behavior of a collection that does not allow duplicate elements. Therefore, the `add()` method returns

Method	Description
void add(int <i>index</i> , E <i>obj</i>)	Inserts <i>obj</i> into the invoking list at the index passed in <i>index</i> . Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
boolean addAll(int <i>index</i> , Collection<? extends E> <i>c</i>)	Inserts all elements of <i>c</i> into the invoking list at the index passed in <i>index</i> . Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.
E get(int <i>index</i>)	Returns the object stored at the specified index within the invoking collection.
int indexOf(Object <i>obj</i>)	Returns the index of the first instance of <i>obj</i> in the invoking list. If <i>obj</i> is not an element of the list, -1 is returned.
int lastIndexOf(Object <i>obj</i>)	Returns the index of the last instance of <i>obj</i> in the invoking list. If <i>obj</i> is not an element of the list, -1 is returned.
ListIterator<E> listIterator()	Returns an iterator to the start of the invoking list.
ListIterator<E> listIterator(int <i>index</i>)	Returns an iterator to the invoking list that begins at the specified <i>index</i> .
E remove(int <i>index</i>)	Removes the element at position <i>index</i> from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
default void replaceAll(UnaryOperator<E> <i>opToApply</i>)	Updates each element in the list with the value obtained from the <i>opToApply</i> function. (Added by JDK 8.)
E set(int <i>index</i> , E <i>obj</i>)	Assigns <i>obj</i> to the location specified by <i>index</i> within the invoking list. Returns the old value.
default void sort(Comparator<? super E> <i>comp</i>)	Sorts the list using the comparator specified by <i>comp</i> . (Added by JDK 8.)
List<E> subList(int <i>start</i> , int <i>end</i>)	Returns a list that includes elements from <i>start</i> to <i>end</i> -1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

Table 18-2 The Methods Declared by **List**

false if an attempt is made to add duplicate elements to a set. It does not specify any additional methods of its own. **Set** is a generic interface that has this declaration:

```
interface Set<E>
```

Here, **E** specifies the type of objects that the set will hold.

The SortedSet Interface

The **SortedSet** interface extends **Set** and declares the behavior of a set sorted in ascending order. **SortedSet** is a generic interface that has this declaration:

```
interface SortedSet<E>
```

Here, **E** specifies the type of objects that the set will hold.

In addition to those methods provided by **Set**, the **SortedSet** interface declares the methods summarized in Table 18-3. Several methods throw a **NoSuchElementException** when no items are contained in the invoking set. A **ClassCastException** is thrown when an object is incompatible with the elements in a set. A **NullPointerException** is thrown if an attempt is made to use a **null** object and **null** is not allowed in the set. An **IllegalArgumentException** is thrown if an invalid argument is used.

SortedSet defines several methods that make set processing more convenient. To obtain the first object in the set, call **first()**. To get the last element, use **last()**. You can obtain a subset of a sorted set by calling **subSet()**, specifying the first and last object in the set. If you need the subset that starts with the first element in the set, use **headSet()**. If you want the subset that ends the set, use **tailSet()**.

Method	Description
Comparator<? super E> comparator()	Returns the invoking sorted set's comparator. If the natural ordering is used for this set, null is returned.
E first()	Returns the first element in the invoking sorted set.
SortedSet<E> headSet(E end)	Returns a SortedSet containing those elements less than <i>end</i> that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.
E last()	Returns the last element in the invoking sorted set.
SortedSet<E> subSet(E start, E end)	Returns a SortedSet that includes those elements between <i>start</i> and <i>end</i> -1. Elements in the returned collection are also referenced by the invoking object.
SortedSet<E> tailSet(E start)	Returns a SortedSet that contains those elements greater than or equal to <i>start</i> that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

Table 18-3 The Methods Declared by **SortedSet**

The NavigableSet Interface

The **NavigableSet** interface extends **SortedSet** and declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values. **NavigableSet** is a generic interface that has this declaration:

```
interface NavigableSet<E>
```

Here, **E** specifies the type of objects that the set will hold. In addition to the methods that it inherits from **SortedSet**, **NavigableSet** adds those summarized in Table 18-4. A

Method	Description
E ceiling(E <i>obj</i>)	Searches the set for the smallest element <i>e</i> such that <i>e</i> \geq <i>obj</i> . If such an element is found, it is returned. Otherwise, null is returned.
Iterator<E> descendingIterator()	Returns an iterator that moves from the greatest to least. In other words, it returns a reverse iterator.
NavigableSet<E> descendingSet()	Returns a NavigableSet that is the reverse of the invoking set. The resulting set is backed by the invoking set.
E floor(E <i>obj</i>)	Searches the set for the largest element <i>e</i> such that <i>e</i> \leq <i>obj</i> . If such an element is found, it is returned. Otherwise, null is returned.
NavigableSet<E> headSet(E <i>upperBound</i> , boolean <i>incl</i>)	Returns a NavigableSet that includes all elements from the invoking set that are less than <i>upperBound</i> . If <i>incl</i> is true , then an element equal to <i>upperBound</i> is included. The resulting set is backed by the invoking set.
E higher(E <i>obj</i>)	Searches the set for the largest element <i>e</i> such that <i>e</i> $>$ <i>obj</i> . If such an element is found, it is returned. Otherwise, null is returned.
E lower(E <i>obj</i>)	Searches the set for the largest element <i>e</i> such that <i>e</i> $<$ <i>obj</i> . If such an element is found, it is returned. Otherwise, null is returned.
E pollFirst()	Returns the first element, removing the element in the process. Because the set is sorted, this is the element with the least value. null is returned if the set is empty.
E pollLast()	Returns the last element, removing the element in the process. Because the set is sorted, this is the element with the greatest value. null is returned if the set is empty.
NavigableSet<E> subSet(E <i>lowerBound</i> , boolean <i>lowIncl</i> , E <i>upperBound</i> , boolean <i>highIncl</i>)	Returns a NavigableSet that includes all elements from the invoking set that are greater than <i>lowerBound</i> and less than <i>upperBound</i> . If <i>lowIncl</i> is true , then an element equal to <i>lowerBound</i> is included. If <i>highIncl</i> is true , then an element equal to <i>upperBound</i> is included. The resulting set is backed by the invoking set.
NavigableSet<E> tailSet(E <i>lowerBound</i> , boolean <i>incl</i>)	Returns a NavigableSet that includes all elements from the invoking set that are greater than <i>lowerBound</i> . If <i>incl</i> is true , then an element equal to <i>lowerBound</i> is included. The resulting set is backed by the invoking set.

Table 18-4 The Methods Declared by **NavigableSet**

ClassCastException is thrown when an object is incompatible with the elements in the set. A **NullPointerException** is thrown if an attempt is made to use a **null** object and **null** is not allowed in the set. An **IllegalArgumentException** is thrown if an invalid argument is used.

The Queue Interface

The **Queue** interface extends **Collection** and declares the behavior of a queue, which is often a first-in, first-out list. However, there are types of queues in which the ordering is based upon other criteria. **Queue** is a generic interface that has this declaration:

```
interface Queue<E>
```

Here, **E** specifies the type of objects that the queue will hold. The methods declared by **Queue** are shown in Table 18-5.

Several methods throw a **ClassCastException** when an object is incompatible with the elements in the queue. A **NullPointerException** is thrown if an attempt is made to store a **null** object and **null** elements are not allowed in the queue. An **IllegalArgumentException** is thrown if an invalid argument is used. An **IllegalStateException** is thrown if an attempt is made to add an element to a fixed-length queue that is full. A **NoSuchElementException** is thrown if an attempt is made to remove an element from an empty queue.

Despite its simplicity, **Queue** offers several points of interest. First, elements can only be removed from the head of the queue. Second, there are two methods that obtain and remove elements: **poll()** and **remove()**. The difference between them is that **poll()** returns **null** if the queue is empty, but **remove()** throws an exception. Third, there are two methods, **element()** and **peek()**, that obtain but don't remove the element at the head of the queue. They differ only in that **element()** throws an exception if the queue is empty, but **peek()** returns **null**. Finally, notice that **offer()** only attempts to add an element to a queue. Because some queues have a fixed length and might be full, **offer()** can fail.

Method	Description
E element()	Returns the element at the head of the queue. The element is not removed. It throws NoSuchElementException if the queue is empty.
boolean offer(E obj)	Attempts to add <i>obj</i> to the queue. Returns true if <i>obj</i> was added and false otherwise.
E peek()	Returns the element at the head of the queue. It returns null if the queue is empty. The element is not removed.
E poll()	Returns the element at the head of the queue, removing the element in the process. It returns null if the queue is empty.
E remove()	Removes the element at the head of the queue, returning the element in the process. It throws NoSuchElementException if the queue is empty.

Table 18-5 The Methods Declared by **Queue**

The Deque Interface

The **Deque** interface extends **Queue** and declares the behavior of a double-ended queue. Double-ended queues can function as standard, first-in, first-out queues or as last-in, first-out stacks. **Deque** is a generic interface that has this declaration:

```
interface Deque<E>
```

Here, **E** specifies the type of objects that the deque will hold. In addition to the methods that it inherits from **Queue**, **Deque** adds those methods summarized in Table 18-6. Several

Method	Description
void addFirst(E <i>obj</i>)	Adds <i>obj</i> to the head of the deque. Throws an IllegalStateException if a capacity-restricted deque is out of space.
void addLast(E <i>obj</i>)	Adds <i>obj</i> to the tail of the deque. Throws an IllegalStateException if a capacity-restricted deque is out of space.
Iterator<E> descendingIterator()	Returns an iterator that moves from the tail to the head of the deque. In other words, it returns a reverse iterator.
E getFirst()	Returns the first element in the deque. The object is not removed from the deque. It throws NoSuchElementException if the deque is empty.
E getLast()	Returns the last element in the deque. The object is not removed from the deque. It throws NoSuchElementException if the deque is empty.
boolean offerFirst(E <i>obj</i>)	Attempts to add <i>obj</i> to the head of the deque. Returns true if <i>obj</i> was added and false otherwise. Therefore, this method returns false when an attempt is made to add <i>obj</i> to a full, capacity-restricted deque.
boolean offerLast(E <i>obj</i>)	Attempts to add <i>obj</i> to the tail of the deque. Returns true if <i>obj</i> was added and false otherwise.
E peekFirst()	Returns the element at the head of the deque. It returns null if the deque is empty. The object is not removed.
E peekLast()	Returns the element at the tail of the deque. It returns null if the deque is empty. The object is not removed.
E pollFirst()	Returns the element at the head of the deque, removing the element in the process. It returns null if the deque is empty.
E pollLast()	Returns the element at the tail of the deque, removing the element in the process. It returns null if the deque is empty.
E pop()	Returns the element at the head of the deque, removing it in the process. It throws NoSuchElementException if the deque is empty.

Table 18-6 The Methods Declared by **Deque**

Method	Description
void push(E <i>obj</i>)	Adds <i>obj</i> to the head of the deque. Throws an IllegalStateException if a capacity-restricted deque is out of space.
E removeFirst()	Returns the element at the head of the deque, removing the element in the process. It throws NoSuchElementException if the deque is empty.
boolean removeFirstOccurrence(Object <i>obj</i>)	Removes the first occurrence of <i>obj</i> from the deque. Returns true if successful and false if the deque did not contain <i>obj</i> .
E removeLast()	Returns the element at the tail of the deque, removing the element in the process. It throws NoSuchElementException if the deque is empty.
boolean removeLastOccurrence(Object <i>obj</i>)	Removes the last occurrence of <i>obj</i> from the deque. Returns true if successful and false if the deque did not contain <i>obj</i> .

Table 18-6 The Methods Declared by **Deque** (continued)

methods throw a **ClassCastException** when an object is incompatible with the elements in the deque. A **NullPointerException** is thrown if an attempt is made to store a **null** object and **null** elements are not allowed in the deque. An **IllegalArgumentException** is thrown if an invalid argument is used. An **IllegalStateException** is thrown if an attempt is made to add an element to a fixed-length deque that is full. A **NoSuchElementException** is thrown if an attempt is made to remove an element from an empty deque.

Notice that **Deque** includes the methods **push()** and **pop()**. These methods enable a **Deque** to function as a stack. Also, notice the **descendingIterator()** method. It returns an iterator that returns elements in reverse order. In other words, it returns an iterator that moves from the end of the collection to the start. A **Deque** implementation can be *capacity-restricted*, which means that only a limited number of elements can be added to the deque. When this is the case, an attempt to add an element to the deque can fail. **Deque** allows you to handle such a failure in two ways. First, methods such as **addFirst()** and **addLast()** throw an **IllegalStateException** if a capacity-restricted deque is full. Second, methods such as **offerFirst()** and **offerLast()** return **false** if the element cannot be added.

The Collection Classes

Now that you are familiar with the collection interfaces, you are ready to examine the standard classes that implement them. Some of the classes provide full implementations that can be used as-is. Others are abstract, providing skeletal implementations that are used as starting points for creating concrete collections. As a general rule, the collection classes are not synchronized, but as you will see later in this chapter, it is possible to obtain synchronized versions.

The core collection classes are summarized in the following table:

Class	Description
AbstractCollection	Implements most of the Collection interface.
AbstractList	Extends AbstractCollection and implements most of the List interface.
AbstractQueue	Extends AbstractCollection and implements parts of the Queue interface.
AbstractSequentialList	Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.
LinkedList	Implements a linked list by extending AbstractSequentialList .
ArrayList	Implements a dynamic array by extending AbstractList .
ArrayDeque	Implements a dynamic double-ended queue by extending AbstractCollection and implementing the Deque interface.
AbstractSet	Extends AbstractCollection and implements most of the Set interface.
EnumSet	Extends AbstractSet for use with enum elements.
HashSet	Extends AbstractSet for use with a hash table.
LinkedHashSet	Extends HashSet to allow insertion-order iterations.
PriorityQueue	Extends AbstractQueue to support a priority-based queue.
TreeSet	Implements a set stored in a tree. Extends AbstractSet .

The following sections examine the concrete collection classes and illustrate their use.

NOTE In addition to the collection classes, several legacy classes, such as **Vector**, **Stack**, and **Hashtable**, have been reengineered to support collections. These are examined later in this chapter.

The **ArrayList** Class

The **ArrayList** class extends **AbstractList** and implements the **List** interface. **ArrayList** is a generic class that has this declaration:

```
class ArrayList<E>
```

Here, **E** specifies the type of objects that the list will hold.

ArrayList supports dynamic arrays that can grow as needed. In Java, standard arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold. But, sometimes, you may not know until run time precisely how large an array you need. To handle this situation, the Collections Framework defines **ArrayList**. In essence, an **ArrayList** is a variable-length array of object references. That is, an **ArrayList** can dynamically increase or decrease in size. Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk.

NOTE Dynamic arrays are also supported by the legacy class **Vector**, which is described later in this chapter.

ArrayList has the constructors shown here:

```
ArrayList()
ArrayList(Collection<? extends E> c)
ArrayList(int capacity)
```

The first constructor builds an empty array list. The second constructor builds an array list that is initialized with the elements of the collection *c*. The third constructor builds an array list that has the specified initial *capacity*. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

The following program shows a simple use of **ArrayList**. An array list is created for objects of type **String**, and then several strings are added to it. (Recall that a quoted string is translated into a **String** object.) The list is then displayed. Some of the elements are removed and the list is displayed again.

```
// Demonstrate ArrayList.
import java.util.*;

class ArrayListDemo {
    public static void main(String args[]) {
        // Create an array list.
        ArrayList<String> al = new ArrayList<String>();

        System.out.println("Initial size of al: " +
                           al.size());

        // Add elements to the array list.
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
        al.add(1, "A2");

        System.out.println("Size of al after additions: " +
                           al.size());

        // Display the array list.
        System.out.println("Contents of al: " + al);

        // Remove elements from the array list.
        al.remove("F");
        al.remove(2);

        System.out.println("Size of al after deletions: " +
                           al.size());

        System.out.println("Contents of al: " + al);
    }
}
```

The output from this program is shown here:

```
Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
```

Notice that `al` starts out empty and grows as elements are added to it. When elements are removed, its size is reduced.

In the preceding example, the contents of a collection are displayed using the default conversion provided by `toString()`, which was inherited from `AbstractCollection`. Although it is sufficient for short, sample programs, you seldom use this method to display the contents of a real-world collection. Usually, you provide your own output routines. But, for the next few examples, the default output created by `toString()` is sufficient.

Although the capacity of an `ArrayList` object increases automatically as objects are stored in it, you can increase the capacity of an `ArrayList` object manually by calling `ensureCapacity()`. You might want to do this if you know in advance that you will be storing many more items in the collection than it can currently hold. By increasing its capacity once, at the start, you can prevent several reallocations later. Because reallocations are costly in terms of time, preventing unnecessary ones improves performance. The signature for `ensureCapacity()` is shown here:

```
void ensureCapacity(int cap)
```

Here, `cap` specifies the new minimum capacity of the collection.

Conversely, if you want to reduce the size of the array that underlies an `ArrayList` object so that it is precisely as large as the number of items that it is currently holding, call `trimToSize()`, shown here:

```
void trimToSize()
```

Obtaining an Array from an `ArrayList`

When working with `ArrayList`, you will sometimes want to obtain an actual array that contains the contents of the list. You can do this by calling `toArray()`, which is defined by `Collection`. Several reasons exist why you might want to convert a collection into an array, such as:

- To obtain faster processing times for certain operations
- To pass an array to a method that is not overloaded to accept a collection
- To integrate collection-based code with legacy code that does not understand collections

Whatever the reason, converting an `ArrayList` to an array is a trivial matter.

As explained earlier, there are two versions of `toArray()`, which are shown again here for your convenience:

```
object[ ] toArray()
<T> T[ ] toArray(T array[ ]) 
```

The first returns an array of **Object**. The second returns an array of elements that have the same type as **T**. Normally, the second form is more convenient because it returns the proper type of array. The following program demonstrates its use:

```
// Convert an ArrayList into an array.  
import java.util.*;  
  
class ArrayListToArray {  
    public static void main(String args[]) {  
        // Create an array list.  
        ArrayList<Integer> al = new ArrayList<Integer>();  
  
        // Add elements to the array list.  
        al.add(1);  
        al.add(2);  
        al.add(3);  
        al.add(4);  
  
        System.out.println("Contents of al: " + al);  
  
        // Get the array.  
        Integer ia[] = new Integer[al.size()];  
        ia = al.toArray(ia);  
  
        int sum = 0;  
  
        // Sum the array.  
        for(int i : ia) sum += i;  
  
        System.out.println("Sum is: " + sum);  
    }  
}
```

The output from the program is shown here:

```
Contents of al: [1, 2, 3, 4]  
Sum is: 10
```

The program begins by creating a collection of integers. Next, **toArray()** is called and it obtains an array of **Integers**. Then, the contents of that array are summed by use of a for-each style **for** loop.

There is something else of interest in this program. As you know, collections can store only references, not values of primitive types. However, autoboxing makes it possible to pass values of type **int** to **add()** without having to manually wrap them within an **Integer**, as the program shows. Autoboxing causes them to be automatically wrapped. In this way, autoboxing significantly improves the ease with which collections can be used to store primitive values.

The LinkedList Class

The **LinkedList** class extends **AbstractSequentialList** and implements the **List**, **Deque**, and **Queue** interfaces. It provides a linked-list data structure. **LinkedList** is a generic class that has this declaration:

```
class LinkedList<E>
```

Here, **E** specifies the type of objects that the list will hold. **LinkedList** has the two constructors shown here:

```
LinkedList()
LinkedList(Collection<? extends E> c)
```

The first constructor builds an empty linked list. The second constructor builds a linked list that is initialized with the elements of the collection *c*.

Because **LinkedList** implements the **Deque** interface, you have access to the methods defined by **Deque**. For example, to add elements to the start of a list, you can use **addFirst()** or **offerFirst()**. To add elements to the end of the list, use **addLast()** or **offerLast()**. To obtain the first element, you can use **getFirst()** or **peekFirst()**. To obtain the last element, use **getLast()** or **peekLast()**. To remove the first element, use **removeFirst()** or **pollFirst()**. To remove the last element, use **removeLast()** or **pollLast()**.

The following program illustrates **LinkedList**:

```
// Demonstrate LinkedList.
import java.util.*;

class LinkedListDemo {
    public static void main(String args[]) {
        // Create a linked list.
        LinkedList<String> ll = new LinkedList<String>();

        // Add elements to the linked list.
        ll.add("F");
        ll.add("B");
        ll.add("D");
        ll.add("E");
        ll.add("C");
        ll.addLast("Z");
        ll.addFirst("A");

        ll.add(1, "A2");

        System.out.println("Original contents of ll: " + ll);

        // Remove elements from the linked list.
        ll.remove("F");
        ll.remove(2);

        System.out.println("Contents of ll after deletion: "
            + ll);
    }
}
```

```

// Remove first and last elements.
ll.removeFirst();
ll.removeLast();

System.out.println("ll after deleting first and last: "
+ ll);

// Get and set a value.

String val = ll.get(2);
ll.set(2, val + " Changed");

System.out.println("ll after change: " + ll);
}
}

```

The output from this program is shown here:

```

Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
ll after deleting first and last: [A2, D, E, C]
ll after change: [A2, D, E Changed, C]

```

Because **LinkedList** implements the **List** interface, calls to **add(E)** append items to the end of the list, as do calls to **addLast()**. To insert items at a specific location, use the **add(int, E)** form of **add()**, as illustrated by the call to **add(1, "A2")** in the example.

Notice how the third element in **ll** is changed by employing calls to **get()** and **set()**. To obtain the current value of an element, pass **get()** the index at which the element is stored. To assign a new value to that index, pass **set()** the index and its new value.

The HashSet Class

HashSet extends **AbstractSet** and implements the **Set** interface. It creates a collection that uses a hash table for storage. **HashSet** is a generic class that has this declaration:

```
class HashSet<E>
```

Here, **E** specifies the type of objects that the set will hold.

As most readers likely know, a hash table stores information by using a mechanism called hashing. In *hashing*, the informational content of a key is used to determine a unique value, called its *hash code*. The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically—you never see the hash code itself. Also, your code can't directly index the hash table. The advantage of hashing is that it allows the execution time of **add()**, **contains()**, **remove()**, and **size()** to remain constant even for large sets.

The following constructors are defined:

```

HashSet()
HashSet(Collection<? extends E> c)
HashSet(int capacity)
HashSet(int capacity, float fillRatio)

```

The first form constructs a default hash set. The second form initializes the hash set by using the elements of *c*. The third form initializes the capacity of the hash set to *capacity*. (The default capacity is 16.) The fourth form initializes both the capacity and the fill ratio (also called *load capacity*) of the hash set from its arguments. The fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded. For constructors that do not take a fill ratio, 0.75 is used.

HashSet does not define any additional methods beyond those provided by its superclasses and interfaces.

It is important to note that **HashSet** does not guarantee the order of its elements, because the process of hashing doesn't usually lend itself to the creation of sorted sets. If you need sorted storage, then another collection, such as **TreeSet**, is a better choice.

Here is an example that demonstrates **HashSet**:

```
// Demonstrate HashSet.
import java.util.*;

class HashSetDemo {
    public static void main(String args[]) {
        // Create a hash set.
        HashSet<String> hs = new HashSet<String>();

        // Add elements to the hash set.
        hs.add("Beta");
        hs.add("Alpha");
        hs.add("Eta");
        hs.add("Gamma");
        hs.add("Epsilon");
        hs.add("Omega");

        System.out.println(hs);
    }
}
```

The following is the output from this program:

```
[Gamma, Eta, Alpha, Epsilon, Omega, Beta]
```

As explained, the elements are not stored in sorted order, and the precise output may vary.

The **LinkedHashSet** Class

The **LinkedHashSet** class extends **HashSet** and adds no members of its own. It is a generic class that has this declaration:

```
class LinkedHashSet<E>
```

Here, **E** specifies the type of objects that the set will hold. Its constructors parallel those in **HashSet**.

LinkedHashSet maintains a linked list of the entries in the set, in the order in which they were inserted. This allows insertion-order iteration over the set. That is, when cycling through a **LinkedHashSet** using an iterator, the elements will be returned in the order in which they were inserted. This is also the order in which they are contained in the string returned by **toString()** when called on a **LinkedHashSet** object. To see the effect of **LinkedHashSet**, try substituting **LinkedHashSet** for **HashSet** in the preceding program. The output will be

```
[Beta, Alpha, Eta, Gamma, Epsilon, Omega]
```

which is the order in which the elements were inserted.

The TreeSet Class

TreeSet extends **AbstractSet** and implements the **NavigableSet** interface. It creates a collection that uses a tree for storage. Objects are stored in sorted, ascending order. Access and retrieval times are quite fast, which makes **TreeSet** an excellent choice when storing large amounts of sorted information that must be found quickly.

TreeSet is a generic class that has this declaration:

```
class TreeSet<E>
```

Here, **E** specifies the type of objects that the set will hold.

TreeSet has the following constructors:

```
TreeSet()
TreeSet(Collection<? extends E> c)
TreeSet(Comparator<? super E> comp)
TreeSet(SortedSet<E> ss)
```

The first form constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements. The second form builds a tree set that contains the elements of *c*. The third form constructs an empty tree set that will be sorted according to the comparator specified by *comp*. (Comparators are described later in this chapter.) The fourth form builds a tree set that contains the elements of *ss*.

Here is an example that demonstrates a **TreeSet**:

```
// Demonstrate TreeSet.
import java.util.*;

class TreeSetDemo {
    public static void main(String args[]) {
        // Create a tree set.
        TreeSet<String> ts = new TreeSet<String>();

        // Add elements to the tree set.
        ts.add("C");
        ts.add("A");
        ts.add("B");
        ts.add("E");
        ts.add("F");
        ts.add("D");
```

```
        System.out.println(ts);
    }
}
```

The output from this program is shown here:

```
[A, B, C, D, E, F]
```

As explained, because **TreeSet** stores its elements in a tree, they are automatically arranged in sorted order, as the output confirms.

Because **TreeSet** implements the **NavigableSet** interface, you can use the methods defined by **NavigableSet** to retrieve elements of a **TreeSet**. For example, assuming the preceding program, the following statement uses **subSet()** to obtain a subset of **ts** that contains the elements between **C** (inclusive) and **F** (exclusive). It then displays the resulting set.

```
System.out.println(ts.subSet("C", "F"));
```

The output from this statement is shown here:

```
[C, D, E]
```

You might want to experiment with the other methods defined by **NavigableSet**.

The PriorityQueue Class

PriorityQueue extends **AbstractQueue** and implements the **Queue** interface. It creates a queue that is prioritized based on the queue's comparator. **PriorityQueue** is a generic class that has this declaration:

```
class PriorityQueue<E>
```

Here, **E** specifies the type of objects stored in the queue. **PriorityQueues** are dynamic, growing as necessary.

PriorityQueue defines the six constructors shown here:

```
PriorityQueue()
PriorityQueue(int capacity)
PriorityQueue(Comparator<? super E> comp) (Added by JDK 8.)
PriorityQueue(int capacity, Comparator<? super E> comp)
PriorityQueue(Collection<? extends E> c)
PriorityQueue(PriorityQueue<? extends E> c)
PriorityQueue(SortedSet<? extends E> c)
```

The first constructor builds an empty queue. Its starting capacity is 11. The second constructor builds a queue that has the specified initial capacity. The third constructor specifies a comparator, and the fourth builds a queue with the specified capacity and comparator. The last three constructors create queues that are initialized with the elements of the collection passed in *c*. In all cases, the capacity grows automatically as elements are added.

If no comparator is specified when a **PriorityQueue** is constructed, then the default comparator for the type of data stored in the queue is used. The default comparator will order the queue in ascending order. Thus, the head of the queue will be the smallest value. However, by providing a custom comparator, you can specify a different ordering scheme. For example, when storing items that include a time stamp, you could prioritize the queue such that the oldest items are first in the queue.

You can obtain a reference to the comparator used by a **PriorityQueue** by calling its **comparator()** method, shown here:

```
Comparator<? super E> comparator()
```

It returns the comparator. If natural ordering is used for the invoking queue, **null** is returned.

One word of caution: Although you can iterate through a **PriorityQueue** using an iterator, the order of that iteration is undefined. To properly use a **PriorityQueue**, you must call methods such as **offer()** and **poll()**, which are defined by the **Queue** interface.

The **ArrayDeque** Class

The **ArrayDeque** class extends **AbstractCollection** and implements the **Deque** interface. It adds no methods of its own. **ArrayDeque** creates a dynamic array and has no capacity restrictions. (The **Deque** interface supports implementations that restrict capacity, but does not require such restrictions.) **ArrayDeque** is a generic class that has this declaration:

```
class ArrayDeque<E>
```

Here, **E** specifies the type of objects stored in the collection.

ArrayDeque defines the following constructors:

```
ArrayDeque()
ArrayDeque(int size)
ArrayDeque(Collection<? extends E> c)
```

The first constructor builds an empty deque. Its starting capacity is 16. The second constructor builds a deque that has the specified initial capacity. The third constructor creates a deque that is initialized with the elements of the collection passed in *c*. In all cases, the capacity grows as needed to handle the elements added to the deque.

The following program demonstrates **ArrayDeque** by using it to create a stack:

```
// Demonstrate ArrayDeque.
import java.util.*;

class ArrayDequeDemo {
    public static void main(String args[]) {
        // Create an array deque.
        ArrayDeque<String> adq = new ArrayDeque<String>();

        // Use an ArrayDeque like a stack.
        adq.push("A");
        adq.push("B");
        adq.push("D");
    }
}
```

```
adq.push("E");
adq.push("F");

System.out.print("Popping the stack: ");
while(adq.peek() != null)
    System.out.print(adq.pop() + " ");
System.out.println();
}
```

The output is shown here:

```
Popping the stack: F E D B A
```

The EnumSet Class

EnumSet extends **AbstractSet** and implements **Set**. It is specifically for use with elements of an **enum** type. It is a generic class that has this declaration:

```
class EnumSet<E extends Enum<E>>
```

Here, **E** specifies the elements. Notice that **E** must extend **Enum<E>**, which enforces the requirement that the elements must be of the specified **enum** type.

EnumSet defines no constructors. Instead, it uses the factory methods shown in Table 18-7 to create objects. All methods can throw **NullPointerException**. The **copyOf()** and **range()** methods can also throw **IllegalArgumentException**. Notice that the **of()** method is overloaded a number of times. This is in the interest of efficiency. Passing a known number of arguments can be faster than using a vararg parameter when the number of arguments is small.

Accessing a Collection via an Iterator

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element. One way to do this is to employ an *iterator*, which is an object that implements either the **Iterator** or the **ListIterator** interface. **Iterator** enables you to cycle through a collection, obtaining or removing elements. **ListIterator** extends **Iterator** to allow bidirectional traversal of a list, and the modification of elements. **Iterator** and **ListIterator** are generic interfaces which are declared as shown here:

```
interface Iterator<E>
interface ListIterator<E>
```

Here, **E** specifies the type of objects being iterated. The **Iterator** interface declares the methods shown in Table 18-8. The methods declared by **ListIterator** (along with those inherited from **Iterator**) are shown in Table 18-9. In both cases, operations that modify the underlying collection are optional. For example, **remove()** will throw **UnsupportedOperationException** when used with a read-only collection. Various other exceptions are possible.

Method	Description
static <E extends Enum<E>> EnumSet<E> allOf(Class<E> <i>t</i>)	Creates an EnumSet that contains the elements in the enumeration specified by <i>t</i> .
static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> <i>e</i>)	Creates an EnumSet that is comprised of those elements not stored in <i>e</i> .
static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> <i>c</i>)	Creates an EnumSet from the elements stored in <i>c</i> .
static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> <i>c</i>)	Creates an EnumSet from the elements stored in <i>c</i> .
static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> <i>t</i>)	Creates an EnumSet that contains the elements that are not in the enumeration specified by <i>t</i> , which is an empty set by definition.
static <E extends Enum<E>> EnumSet<E> of(E <i>v</i> , E ... <i>varargs</i>)	Creates an EnumSet that contains <i>v</i> and zero or more additional enumeration values.
static <E extends Enum<E>> EnumSet<E> of(E <i>v</i>)	Creates an EnumSet that contains <i>v</i> .
static <E extends Enum<E>> EnumSet<E> of(E <i>v1</i> , E <i>v2</i>)	Creates an EnumSet that contains <i>v1</i> and <i>v2</i> .
static <E extends Enum<E>> EnumSet<E> of(E <i>v1</i> , E <i>v2</i> , E <i>v3</i>)	Creates an EnumSet that contains <i>v1</i> through <i>v3</i> .
static <E extends Enum<E>> EnumSet<E> of(E <i>v1</i> , E <i>v2</i> , E <i>v3</i> , E <i>v4</i>)	Creates an EnumSet that contains <i>v1</i> through <i>v4</i> .
static <E extends Enum<E>> EnumSet<E> of(E <i>v1</i> , E <i>v2</i> , E <i>v3</i> , E <i>v4</i> , E <i>v5</i>)	Creates an EnumSet that contains <i>v1</i> through <i>v5</i> .
static <E extends Enum<E>> EnumSet<E> range(E <i>start</i> , E <i>end</i>)	Creates an EnumSet that contains the elements in the range specified by <i>start</i> and <i>end</i> .

Table 18-7 The Methods Declared by **EnumSet**

Method	Description
default void forEachRemaining(Consumer<? super E> <i>action</i>)	The action specified by <i>action</i> is executed on each unprocessed element in the collection. (Added by JDK 8.)
boolean hasNext()	Returns true if there are more elements. Otherwise, returns false .
E next()	Returns the next element. Throws NoSuchElementException if there is not a next element.
default void remove()	Removes the current element. Throws IllegalStateException if an attempt is made to call remove() that is not preceded by a call to next() . The default version throws an UnsupportedOperationException .

Table 18-8 The Methods Declared by **Iterator**

Method	Description
void add(E <i>obj</i>)	Inserts <i>obj</i> into the list in front of the element that will be returned by the next call to next() .
default void forEachRemaining(Consumer<? super E> <i>action</i>)	The action specified by <i>action</i> is executed on each unprocessed element in the collection. (Added by JDK 8.)
boolean hasNext()	Returns true if there is a next element. Otherwise, returns false .
boolean hasPrevious()	Returns true if there is a previous element. Otherwise, returns false .
E next()	Returns the next element. A NoSuchElementException is thrown if there is not a next element.
int nextIndex()	Returns the index of the next element. If there is not a next element, returns the size of the list.
E previous()	Returns the previous element. A NoSuchElementException is thrown if there is not a previous element.
int previousIndex()	Returns the index of the previous element. If there is not a previous element, returns -1.
void remove()	Removes the current element from the list. An IllegalStateException is thrown if remove() is called before next() or previous() is invoked.
void set(E <i>obj</i>)	Assigns <i>obj</i> to the current element. This is the element last returned by a call to either next() or previous() .

Table 18-9 The Methods Provided by **ListIterator**

NOTE Beginning with JDK 8, you can also use a **Spliterator** to cycle through a collection. **Spliterator** works differently than does **Iterator**, and it is described later in this chapter.

Using an Iterator

Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an **iterator()** method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time. In general, to use an iterator to cycle through the contents of a collection, follow these steps:

1. Obtain an iterator to the start of the collection by calling the collection's **iterator()** method.
2. Set up a loop that makes a call to **hasNext()**. Have the loop iterate as long as **hasNext()** returns **true**.
3. Within the loop, obtain each element by calling **next()**.

For collections that implement **List**, you can also obtain an iterator by calling **listIterator()**. As explained, a list iterator gives you the ability to access the collection in either the forward or backward direction and lets you modify an element. Otherwise, **ListIterator** is used just like **Iterator**.

The following example implements these steps, demonstrating both the **Iterator** and **ListIterator** interfaces. It uses an **ArrayList** object, but the general principles apply to any type of collection. Of course, **ListIterator** is available only to those collections that implement the **List** interface.

```
// Demonstrate iterators.
import java.util.*;

class IteratorDemo {
    public static void main(String args[]) {
        // Create an array list.
        ArrayList<String> al = new ArrayList<String>();

        // Add elements to the array list.
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");

        // Use iterator to display contents of al.
        System.out.print("Original contents of al: ");
        Iterator<String> itr = al.iterator();
        while(itr.hasNext()) {
            String element = itr.next();
            System.out.print(element + " ");
        }
        System.out.println();

        // Modify objects being iterated.
        ListIterator<String> litr = al.listIterator();
        while(litr.hasNext()) {
            String element = litr.next();
            litr.set(element + "+");
        }

        System.out.print("Modified contents of al: ");
        itr = al.iterator();
        while(itr.hasNext()) {
            String element = itr.next();
            System.out.print(element + " ");
        }
        System.out.println();

        // Now, display the list backwards.
        System.out.print("Modified list backwards: ");
        while(litr.hasPrevious()) {
```

```
        String element = litr.previous();
        System.out.print(element + " ");
    }
    System.out.println();
}
}
```

The output is shown here:

```
Original contents of al: C A E B D F
Modified contents of al: C+ A+ E+ B+ D+ F+
Modified list backwards: F+ D+ B+ E+ A+ C+
```

Pay special attention to how the list is displayed in reverse. After the list is modified, `litr` points to the end of the list. (Remember, `litr.hasNext()` returns `false` when the end of the list has been reached.) To traverse the list in reverse, the program continues to use `litr`, but this time it checks to see whether it has a previous element. As long as it does, that element is obtained and displayed.

The For-Each Alternative to Iterators

If you won't be modifying the contents of a collection or obtaining elements in reverse order, then the for-each version of the `for` loop is often a more convenient alternative to cycling through a collection than is using an iterator. Recall that the `for` can cycle through any collection of objects that implement the **Iterable** interface. Because all of the collection classes implement this interface, they can all be operated upon by the `for`.

The following example uses a `for` loop to sum the contents of a collection:

```
// Use the for-each for loop to cycle through a collection.
import java.util.*;

class ForEachDemo {
    public static void main(String args[]) {
        // Create an array list for integers.
        ArrayList<Integer> vals = new ArrayList<Integer>();

        // Add values to the array list.
        vals.add(1);
        vals.add(2);
        vals.add(3);
        vals.add(4);
        vals.add(5);

        // Use for loop to display the values.
        System.out.print("Contents of vals: ");
        for(int v : vals)
            System.out.print(v + " ");

        System.out.println();

        // Now, sum the values by using a for loop.
    }
}
```

```

        int sum = 0;
        for(int v : vals)
            sum += v;

        System.out.println("Sum of values: " + sum);
    }
}

```

The output from the program is shown here:

```

Contents of vals: 1 2 3 4 5
Sum of values: 15

```

As you can see, the **for** loop is substantially shorter and simpler to use than the iterator-based approach. However, it can only be used to cycle through a collection in the forward direction, and you can't modify the contents of the collection.

Spliterators

JDK 8 adds a new type of iterator called a *spliterator* that is defined by the **Spliterator** interface. A spliterator cycles through a sequence of elements, and in this regard, it is similar to the iterators just described. However, the techniques required to use it differ. Furthermore, it offers substantially more functionality than does either **Iterator** or **ListIterator**. Perhaps the most important aspect of **Spliterator** is its ability to provide support for parallel iteration of portions of the sequence. Thus, **Spliterator** supports parallel programming. (See Chapter 28 for information on concurrency and parallel programming.) However, you can use **Spliterator** even if you won't be using parallel execution. One reason you might want to do so is because it offers a streamlined approach that combines the *hasNext* and *next* operations into one method.

Spliterator is a generic interface that is declared like this:

```
interface Spliterator<T>
```

Here, **T** is the type of elements being iterated. **Spliterator** declares the methods shown in Table 18-10.

Using **Spliterator** for basic iteration tasks is quite easy: simply call **tryAdvance()** until it returns **false**. If you will be applying the same action to each element in the sequence, **forEachRemaining()** offers a streamlined alternative. In both cases, the action that will occur with each iteration is defined by what the **Consumer** object does with each element. **Consumer** is a functional interface that applies an action to an object. It is a generic functional interface declared in **java.util.function**. (See Chapter 19 for information on **java.util.function**.) **Consumer** specifies only one abstract method, **accept()**, which is shown here:

```
void accept(T objRef)
```

In the case of **tryAdvance()**, each iteration passes the next element in the sequence to *objRef*. Often, the easiest way to implement **Consumer** is by use of a lambda expression.

Method	Description
int characteristics()	Returns the characteristics of the invoking spliterator, encoded into an integer.
long estimateSize()	Estimates the number of elements left to iterate and returns the result. Returns Long.MAX_VALUE if the count cannot be obtained for any reason.
default void forEachRemaining(Consumer<? super T> action)	Applies <i>action</i> to each unprocessed element in the data source.
default Comparator<? super T> getComparator()	Returns the comparator used by the invoking spliterator or null if natural ordering is used. If the sequence is unordered, IllegalStateException is thrown.
default long getExactSizeIfKnown()	If the invoking spliterator is sized, returns the number of elements left to iterate. Returns -1 otherwise.
default boolean hasCharacteristics(int val)	Returns true if the invoking spliterator has the characteristics passed in <i>val</i> . Returns false otherwise.
boolean tryAdvance(Consumer<? super T> action)	Executes <i>action</i> on the next element in the iteration. Returns true if there is a next element. Returns false if no elements remain.
Spliterator<T> trySplit()	If possible, splits the invoking spliterator, returning a reference to a new spliterator for the partition. Otherwise, returns null . Thus, if successful, the original spliterator iterates over one portion of the sequence and the returned spliterator iterates over the other portion.

Table 18-10 The Methods Declared by **Spliterator**

The following program provides a simple example of **Spliterator**. Notice that the program demonstrates both **tryAdvance()** and **forEachRemaining()**. Also notice how these methods combine the actions of **Iterator's** **next()** and **hasNext()** methods into a single call.

```
// A simple Spliterator demonstration.
import java.util.*;

class SpliteratorDemo {

    public static void main(String args[]) {
        // Create an array list for doubles.
        ArrayList<Double> vals = new ArrayList<>();

        // Add values to the array list.
        vals.add(1.0);
        vals.add(2.0);
        vals.add(3.0);
        vals.add(4.0);
        vals.add(5.0);

        // Print the values.
        for (Double val : vals) {
            System.out.println(val);
        }
    }
}
```

```

// Use tryAdvance() to display contents of vals.
System.out.print("Contents of vals:\n");
Spliterator<Double> splitr = valsspliterator();
while(splitr.tryAdvance((n) -> System.out.println(n)));
System.out.println();

// Create new list that contains square roots.
splitr = valsspliterator();
ArrayList<Double> sqrs = new ArrayList<>();
while(splitr.tryAdvance((n) -> sqrs.add(Math.sqrt(n))));

// Use forEachRemaining() to display contents of sqrs.
System.out.print("Contents of sqrs:\n");
splitr = sqrsspliterator();
splitr.forEachRemaining((n) -> System.out.println(n));
System.out.println();
}
}

```

The output is shown here:

Contents of vals:

1.0
2.0
3.0
4.0
5.0

Contents of sqrs:

1.0
1.4142135623730951
1.7320508075688772
2.0
2.23606797749979

Although this program demonstrates the mechanics of using **Spliterator**, it does not reveal its full power. As mentioned, **Spliterator**'s maximum benefit is found in situations that involve parallel processing.

In Table 18-10, notice the methods **characteristics()** and **hasCharacteristics()**. Each **Spliterator** has a set of attributes, called *characteristics*, associated with it. These are defined by static **int** fields in **Spliterator**, such as **SORTED**, **DISTINCT**, **SIZED**, and **IMMUTABLE**, to name a few. You can obtain the characteristics by calling **characteristics()**. You can determine if a characteristic is present by calling **hasCharacteristics()**. Often, you won't need to access a **Spliterator**'s characteristics, but in some cases, they can aid in creating efficient, resilient code.

NOTE For a further discussion of **Spliterator**, see Chapter 29, where it is used in the context of the new stream API. For a discussion of lambda expressions, see Chapter 15. See Chapter 28 for a discussion of parallel programming and concurrency.

There are several nested subinterfaces of **Spliterator** designed for use with the primitive types **double**, **int**, and **long**. These are called **Spliterator.OfDouble**, **Spliterator.OfInt**, and **Spliterator.OfLong**. There is also a generalized version called **Spliterator.OfPrimitive()**, which offers additional flexibility and serves as a superinterface of the aforementioned ones.

Storing User-Defined Classes in Collections

For the sake of simplicity, the foregoing examples have stored built-in objects, such as **String** or **Integer**, in a collection. Of course, collections are not limited to the storage of built-in objects. Quite the contrary. The power of collections is that they can store any type of object, including objects of classes that you create. For example, consider the following example that uses a **LinkedList** to store mailing addresses:

```
// Display the mailing list.  
for(Address element : ml)  
    System.out.println(element + "\n");  
  
    System.out.println();  
}  
}
```

The output from the program is shown here:

```
J.W. West  
11 Oak Ave  
Urbana IL 61801
```

```
Ralph Baker  
1142 Maple Lane  
Mahomet IL 61853
```

```
Tom Carlton  
867 Elm St  
Champaign IL 61820
```

Aside from storing a user-defined class in a collection, another important thing to notice about the preceding program is that it is quite short. When you consider that it sets up a linked list that can store, retrieve, and process mailing addresses in about 50 lines of code, the power of the Collections Framework begins to become apparent. As most readers know, if all of this functionality had to be coded manually, the program would be several times longer. Collections offer off-the-shelf solutions to a wide variety of programming problems. You should use them whenever the situation presents itself.

The RandomAccess Interface

The **RandomAccess** interface contains no members. However, by implementing this interface, a collection signals that it supports efficient random access to its elements. Although a collection might support random access, it might not do so efficiently. By checking for the **RandomAccess** interface, client code can determine at run time whether a collection is suitable for certain types of random access operations—especially as they apply to large collections. (You can use **instanceof** to determine if a class implements an interface.) **RandomAccess** is implemented by **ArrayList** and by the legacy **Vector** class, among others.

Working with Maps

A *map* is an object that stores associations between keys and values, or *key/value pairs*. Given a key, you can find its value. Both keys and values are objects. The keys must be unique, but the values may be duplicated. Some maps can accept a **null** key and **null** values, others cannot.

There is one key point about maps that is important to mention at the outset: they don't implement the **Iterable** interface. This means that you *cannot* cycle through a map using a for-each style **for** loop. Furthermore, you can't obtain an iterator to a map. However, as you will soon see, you can obtain a collection-view of a map, which does allow the use of either the **for** loop or an iterator.

The Map Interfaces

Because the map interfaces define the character and nature of maps, this discussion of maps begins with them. The following interfaces support maps:

Interface	Description
Map	Maps unique keys to values.
Map.Entry	Describes an element (a key/value pair) in a map. This is an inner class of Map .
NavigableMap	Extends SortedMap to handle the retrieval of entries based on closest-match searches.
SortedMap	Extends Map so that the keys are maintained in ascending order.

Each interface is examined next, in turn.

The Map Interface

The **Map** interface maps unique keys to values. A *key* is an object that you use to retrieve a value at a later date. Given a key and a value, you can store the value in a **Map** object. After the value is stored, you can retrieve it by using its key. **Map** is generic and is declared as shown here:

```
interface Map<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The methods declared by **Map** are summarized in Table 18-11. Several methods throw a **ClassCastException** when an object is incompatible with the elements in a map. A **NullPointerException** is thrown if an attempt is made to use a **null** object and **null** is not allowed in the map. An **UnsupportedOperationException** is thrown when an attempt is made to change an unmodifiable map. An **IllegalArgumentException** is thrown if an invalid argument is used.

Maps revolve around two basic operations: **get()** and **put()**. To put a value into a map, use **put()**, specifying the key and the value. To obtain a value, call **get()**, passing the key as an argument. The value is returned.

As mentioned earlier, although part of the Collections Framework, maps are not, themselves, collections because they do not implement the **Collection** interface. However, you can obtain a collection-view of a map. To do this, you can use the **entrySet()** method. It returns a **Set** that contains the elements in the map. To obtain a collection-view of the keys, use **keySet()**. To get a collection-view of the values, use **values()**. For all three collection-views, the collection is backed by the map. Changing one affects the other. Collection-views are the means by which maps are integrated into the larger Collections Framework.

Method	Description
void clear()	Removes all key/value pairs from the invoking map.
default V compute(K <i>k</i> , BiFunction<? super K, ? super V, ? extends V> <i>func</i>)	Calls <i>func</i> to construct a new value. If <i>func</i> returns non- null , the new key/value pair is added to the map, any preexisting pairing is removed, and the new value is returned. If <i>func</i> returns null , any preexisting pairing is removed, and null is returned. (Added by JDK 8.)
default V computeIfAbsent(K <i>k</i> , Function<? super K, ? extends V> <i>func</i>)	Returns the value associated with the key <i>k</i> . Otherwise, the value is constructed through a call to <i>func</i> and the pairing is entered into the map and the constructed value is returned. If no value can be constructed, null is returned. (Added by JDK 8.)
default V computeIfPresent(K <i>k</i> , BiFunction<? super K, ? super V, ? extends V> <i>func</i>)	If <i>k</i> is in the map, a new value is constructed through a call to <i>func</i> and the new value replaces the old value in the map. In this case, the new value is returned. If the value returned by <i>func</i> is null , the existing key and value are removed from the map and null is returned. (Added by JDK 8.)
boolean containsKey(Object <i>k</i>)	Returns true if the invoking map contains <i>k</i> as a key. Otherwise, returns false .
boolean containsValue(Object <i>v</i>)	Returns true if the map contains <i>v</i> as a value. Otherwise, returns false .
Set<Map.Entry<K, V>> entrySet()	Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry . Thus, this method provides a set-view of the invoking map.
boolean equals(Object <i>obj</i>)	Returns true if <i>obj</i> is a Map and contains the same entries. Otherwise, returns false .
default void forEach(BiConsumer< ? super K, ? super V> <i>action</i>)	Executes <i>action</i> on each element in the invoking map. A ConcurrentModificationException will be thrown if an element is removed during the process. (Added by JDK 8.)
V get(Object <i>k</i>)	Returns the value associated with the key <i>k</i> . Returns null if the key is not found.
default V getOrDefault(Object <i>k</i> , V <i>defVal</i>)	Returns the value associated with <i>k</i> if it is in the map. Otherwise, <i>defVal</i> is returned. (Added by JDK 8.)
int hashCode()	Returns the hash code for the invoking map.
boolean isEmpty()	Returns true if the invoking map is empty. Otherwise, returns false .
Set<K> keySet()	Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.

Table 18-11 The Methods Declared by **Map**

Method	Description
default V merge(K <i>k</i> , V <i>v</i> , BiFunction<? super V, ? super V, ? extends V> <i>func</i>)	If <i>k</i> is not in the map, the pairing <i>k</i> , <i>v</i> is added to the map. In this case, <i>v</i> is returned. Otherwise, <i>func</i> returns a new value based on the old value, the key is updated to use this value, and merge() returns this value. If the value returned by <i>func</i> is null , the existing key and value are removed from the map and null is returned. (Added by JDK 8.)
V put(K <i>k</i> , V <i>v</i>)	Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are <i>k</i> and <i>v</i> , respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.
void putAll(Map<? extends K, ? extends V> <i>m</i>)	Puts all the entries from <i>m</i> into this map.
default V putIfAbsent(K <i>k</i> , V <i>v</i>)	Inserts the key/value pair into the invoking map if this pairing is not already present or if the existing value is null . Returns the old value. The null value is returned when no previous mapping exists, or the value is null . (Added by JDK 8.)
V remove(Object <i>k</i>)	Removes the entry whose key equals <i>k</i> .
default boolean remove(Object <i>k</i> , Object <i>v</i>)	If the key/value pair specified by <i>k</i> and <i>v</i> is in the invoking map, it is removed and true is returned. Otherwise, false is returned. (Added by JDK 8.)
default boolean replace(K <i>k</i> , V <i>oldV</i> , V <i>newV</i>)	If the key/value pair specified by <i>k</i> and <i>oldV</i> is in the invoking map, the value is replaced by <i>newV</i> and true is returned. Otherwise false is returned. (Added by JDK 8.)
default V replace(K <i>k</i> , V <i>v</i>)	If the key specified by <i>k</i> is in the invoking map, its value is set to <i>v</i> and the previous value is returned. Otherwise, null is returned. (Added by JDK 8.)
default void replaceAll(BiFunction< ? super K, ? super V, ? extends V> <i>func</i>)	Executes <i>func</i> on each element of the invoking map, replacing the element with the result returned by <i>func</i> . A ConcurrentModificationException will be thrown if an element is removed during the process. (Added by JDK 8.)
int size()	Returns the number of key/value pairs in the map.
Collection<V> values()	Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

Table 18-11 The Methods Declared by **Map** (continued)

The SortedMap Interface

The **SortedMap** interface extends **Map**. It ensures that the entries are maintained in ascending order based on the keys. **SortedMap** is generic and is declared as shown here:

```
interface SortedMap<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The methods declared by **SortedMap** are summarized in Table 18-12. Several methods throw a **NoSuchElementException** when no items are in the invoking map. A **ClassCastException** is thrown when an object is incompatible with the elements in a map. A **NullPointerException** is thrown if an attempt is made to use a **null** object when **null** is not allowed in the map. An **IllegalArgumentException** is thrown if an invalid argument is used.

Sorted maps allow very efficient manipulations of *submaps* (in other words, subsets of a map). To obtain a submap, use **headMap()**, **tailMap()**, or **subMap()**. The submap returned by these methods is backed by the invoking map. Changing one changes the other. To get the first key in the set, call **firstKey()**. To get the last key, use **lastKey()**.

The NavigableMap Interface

The **NavigableMap** interface extends **SortedMap** and declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys. **NavigableMap** is a generic interface that has this declaration:

```
interface NavigableMap<K,V>
```

Here, **K** specifies the type of the keys, and **V** specifies the type of the values associated with the keys. In addition to the methods that it inherits from **SortedMap**, **NavigableMap** adds those summarized in Table 18-13. Several methods throw a **ClassCastException** when an object is incompatible with the keys in the map. A **NullPointerException** is thrown if an attempt is made to use a **null** object and **null** keys are not allowed in the set. An **IllegalArgumentException** is thrown if an invalid argument is used.

Method	Description
Comparator<? super K> comparator()	Returns the invoking sorted map's comparator. If natural ordering is used for the invoking map, null is returned.
K firstKey()	Returns the first key in the invoking map.
SortedMap<K, V> headMap(K end)	Returns a sorted map for those map entries with keys that are less than <i>end</i> .
K lastKey()	Returns the last key in the invoking map.
SortedMap<K, V> subMap(K start, K end)	Returns a map containing those entries with keys that are greater than or equal to <i>start</i> and less than <i>end</i> .
SortedMap<K, V> tailMap(K start)	Returns a map containing those entries with keys that are greater than or equal to <i>start</i> .

Table 18-12 The Methods Declared by **SortedMap**

Method	Description
Map.Entry<K,V> ceilingEntry(K <i>obj</i>)	Searches the map for the smallest key <i>k</i> such that <i>k</i> \geq <i>obj</i> . If such a key is found, its entry is returned. Otherwise, null is returned.
K ceilingKey(K <i>obj</i>)	Searches the map for the smallest key <i>k</i> such that <i>k</i> \geq <i>obj</i> . If such a key is found, it is returned. Otherwise, null is returned.
NavigableSet<K> descendingKeySet()	Returns a NavigableSet that contains the keys in the invoking map in reverse order. Thus, it returns a reverse set-view of the keys. The resulting set is backed by the map.
NavigableMap<K,V> descendingMap()	Returns a NavigableMap that is the reverse of the invoking map. The resulting map is backed by the invoking map.
Map.Entry<K,V> firstEntry()	Returns the first entry in the map. This is the entry with the least key.
Map.Entry<K,V> floorEntry(K <i>obj</i>)	Searches the map for the largest key <i>k</i> such that <i>k</i> \leq <i>obj</i> . If such a key is found, its entry is returned. Otherwise, null is returned.
K floorKey(K <i>obj</i>)	Searches the map for the largest key <i>k</i> such that <i>k</i> \leq <i>obj</i> . If such a key is found, it is returned. Otherwise, null is returned.
NavigableMap<K,V> headMap(K <i>upperBound</i> , boolean <i>incl</i>)	Returns a NavigableMap that includes all entries from the invoking map that have keys that are less than <i>upperBound</i> . If <i>incl</i> is true , then an element equal to <i>upperBound</i> is included. The resulting map is backed by the invoking map.
Map.Entry<K,V> higherEntry(K <i>obj</i>)	Searches the set for the largest key <i>k</i> such that <i>k</i> $>$ <i>obj</i> . If such a key is found, its entry is returned. Otherwise, null is returned.
K higherKey(K <i>obj</i>)	Searches the set for the largest key <i>k</i> such that <i>k</i> $>$ <i>obj</i> . If such a key is found, it is returned. Otherwise, null is returned.
Map.Entry<K,V> lastEntry()	Returns the last entry in the map. This is the entry with the largest key.
Map.Entry<K,V> lowerEntry(K <i>obj</i>)	Searches the set for the largest key <i>k</i> such that <i>k</i> $<$ <i>obj</i> . If such a key is found, its entry is returned. Otherwise, null is returned.
K lowerKey(K <i>obj</i>)	Searches the set for the largest key <i>k</i> such that <i>k</i> $<$ <i>obj</i> . If such a key is found, it is returned. Otherwise, null is returned.

Table 18-13 The Methods Declared by **NavigableMap**

Method	Description
<code>NavigableSet<K> navigableKeySet()</code>	Returns a NavigableSet that contains the keys in the invoking map. The resulting set is backed by the invoking map.
<code>Map.Entry<K,V> pollFirstEntry()</code>	Returns the first entry, removing the entry in the process. Because the map is sorted, this is the entry with the least key value. null is returned if the map is empty.
<code>Map.Entry<K,V> pollLastEntry()</code>	Returns the last entry, removing the entry in the process. Because the map is sorted, this is the entry with the greatest key value. null is returned if the map is empty.
<code>NavigableMap<K,V> subMap(K lowerBound, boolean lowIncl, K upperBound boolean highIncl)</code>	Returns a NavigableMap that includes all entries from the invoking map that have keys that are greater than <i>lowerBound</i> and less than <i>upperBound</i> . If <i>lowIncl</i> is true , then an element equal to <i>lowerBound</i> is included. If <i>highIncl</i> is true , then an element equal to <i>highIncl</i> is included. The resulting map is backed by the invoking map.
<code>NavigableMap<K,V> tailMap(K lowerBound, boolean incl)</code>	Returns a NavigableMap that includes all entries from the invoking map that have keys that are greater than <i>lowerBound</i> . If <i>incl</i> is true , then an element equal to <i>lowerBound</i> is included. The resulting map is backed by the invoking map.

Table 18-13 The Methods Declared by **NavigableMap** (*continued*)

The **Map.Entry** Interface

The **Map.Entry** interface enables you to work with a map entry. Recall that the **entrySet()** method declared by the **Map** interface returns a **Set** containing the map entries. Each of these set elements is a **Map.Entry** object. **Map.Entry** is generic and is declared like this:

```
interface Map.Entry<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values. Table 18-14 summarizes the non-static methods declared by **Map.Entry**. JDK 8 adds two static methods. The first is **comparingByKey()**, which returns a **Comparator** that compares entries by key. The second is **comparingByValue()**, which returns a **Comparator** that compares entries by value.

Method	Description
boolean equals(Object <i>obj</i>)	Returns true if <i>obj</i> is a Map.Entry whose key and value are equal to that of the invoking object.
K getKey()	Returns the key for this map entry.
V getValue()	Returns the value for this map entry.
int hashCode()	Returns the hash code for this map entry.
V setValue(V <i>v</i>)	Sets the value for this map entry to <i>v</i> . A ClassCastException is thrown if <i>v</i> is not the correct type for the map. An IllegalArgumentException is thrown if there is a problem with <i>v</i> . A NullPointerException is thrown if <i>v</i> is null and the map does not permit null keys. An UnsupportedOperationException is thrown if the map cannot be changed.

Table 18-14 The Non-Static Methods Declared by **Map.Entry**

The Map Classes

Several classes provide implementations of the map interfaces. The classes that can be used for maps are summarized here:

Class	Description
AbstractMap	Implements most of the Map interface.
EnumMap	Extends AbstractMap for use with enum keys.
HashMap	Extends AbstractMap to use a hash table.
TreeMap	Extends AbstractMap to use a tree.
WeakHashMap	Extends AbstractMap to use a hash table with weak keys.
LinkedHashMap	Extends HashMap to allow insertion-order iterations.
IdentityHashMap	Extends AbstractMap and uses reference equality when comparing documents.

Notice that **AbstractMap** is a superclass for all concrete map implementations.

WeakHashMap implements a map that uses “weak keys,” which allows an element in a map to be garbage-collected when its key is otherwise unused. This class is not discussed further here. The other map classes are described next.

The HashMap Class

The **HashMap** class extends **AbstractMap** and implements the **Map** interface. It uses a hash table to store the map. This allows the execution time of **get()** and **put()** to remain constant even for large sets. **HashMap** is a generic class that has this declaration:

```
class HashMap<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The following constructors are defined:

```
HashMap()
HashMap(Map<? extends K, ? extends V> m)
HashMap(int capacity)
HashMap(int capacity, float fillRatio)
```

The first form constructs a default hash map. The second form initializes the hash map by using the elements of *m*. The third form initializes the capacity of the hash map to *capacity*. The fourth form initializes both the capacity and fill ratio of the hash map by using its arguments. The meaning of capacity and fill ratio is the same as for **HashSet**, described earlier. The default capacity is 16. The default fill ratio is 0.75.

HashMap implements **Map** and extends **AbstractMap**. It does not add any methods of its own.

You should note that a hash map does not guarantee the order of its elements. Therefore, the order in which elements are added to a hash map is not necessarily the order in which they are read by an iterator.

The following program illustrates **HashMap**. It maps names to account balances. Notice how a set-view is obtained and used.

```
import java.util.*;

class HashMapDemo {
    public static void main(String args[]) {

        // Create a hash map.
        HashMap<String, Double> hm = new HashMap<String, Double>();

        // Put elements to the map
        hm.put("John Doe", new Double(3434.34));
        hm.put("Tom Smith", new Double(123.22));
        hm.put("Jane Baker", new Double(1378.00));
        hm.put("Tod Hall", new Double(99.22));
        hm.put("Ralph Smith", new Double(-19.08));

        // Get a set of the entries.
        Set<Map.Entry<String, Double>> set = hm.entrySet();

        // Display the set.
        for(Map.Entry<String, Double> me : set) {
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());
        }

        System.out.println();

        // Deposit 1000 into John Doe's account.
        double balance = hm.get("John Doe");
        hm.put("John Doe", balance + 1000);
    }
}
```

```
        System.out.println("John Doe's new balance: " +
                           hm.get("John Doe")) ;
    }
}
```

Output from this program is shown here (the precise order may vary):

```
Ralph Smith: -19.08
Tom Smith: 123.22
John Doe: 3434.34
Tod Hall: 99.22
Jane Baker: 1378.0

John Doe's new balance: 4434.34
```

The program begins by creating a hash map and then adds the mapping of names to balances. Next, the contents of the map are displayed by using a set-view, obtained by calling `entrySet()`. The keys and values are displayed by calling the `getKey()` and `getValue()` methods that are defined by `Map.Entry`. Pay close attention to how the deposit is made into John Doe's account. The `put()` method automatically replaces any preexisting value that is associated with the specified key with the new value. Thus, after John Doe's account is updated, the hash map will still contain just one "John Doe" account.

The TreeMap Class

The `TreeMap` class extends `AbstractMap` and implements the `NavigableMap` interface. It creates maps stored in a tree structure. A `TreeMap` provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval. You should note that, unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order. `TreeMap` is a generic class that has this declaration:

```
class TreeMap<K, V>
```

Here, `K` specifies the type of keys, and `V` specifies the type of values.

The following `TreeMap` constructors are defined:

```
TreeMap()
TreeMap(Comparator<? super K> comp)
TreeMap(Map<? extends K, ? extends V> m)
TreeMap(SortedMap<K, ? extends V> sm)
```

The first form constructs an empty tree map that will be sorted by using the natural order of its keys. The second form constructs an empty tree-based map that will be sorted by using the `Comparator` `comp`. (Comparators are discussed later in this chapter.) The third form initializes a tree map with the entries from `m`, which will be sorted by using the natural order of the keys. The fourth form initializes a tree map with the entries from `sm`, which will be sorted in the same order as `sm`.

`TreeMap` has no map methods beyond those specified by the `NavigableMap` interface and the `AbstractMap` class.

The following program reworks the preceding example so that it uses **TreeMap**:

```
import java.util.*;

class TreeMapDemo {
    public static void main(String args[]) {

        // Create a tree map.
        TreeMap<String, Double> tm = new TreeMap<String, Double>();

        // Put elements to the map.
        tm.put("John Doe", new Double(3434.34));
        tm.put("Tom Smith", new Double(123.22));
        tm.put("Jane Baker", new Double(1378.00));
        tm.put("Tod Hall", new Double(99.22));
        tm.put("Ralph Smith", new Double(-19.08));

        // Get a set of the entries.
        Set<Map.Entry<String, Double>> set = tm.entrySet();

        // Display the elements.
        for(Map.Entry<String, Double> me : set) {
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());
        }
        System.out.println();

        // Deposit 1000 into John Doe's account.
        double balance = tm.get("John Doe");
        tm.put("John Doe", balance + 1000);

        System.out.println("John Doe's new balance: " +
                           tm.get("John Doe"));
    }
}
```

The following is the output from this program:

```
Jane Baker: 1378.0
John Doe: 3434.34
Ralph Smith: -19.08
Todd Hall: 99.22
Tom Smith: 123.22

John Doe's current balance: 4434.34
```

Notice that **TreeMap** sorts the keys. However, in this case, they are sorted by first name instead of last name. You can alter this behavior by specifying a comparator when the map is created, as described shortly.

The **LinkedHashMap** Class

LinkedHashMap extends **HashMap**. It maintains a linked list of the entries in the map, in the order in which they were inserted. This allows insertion-order iteration over the map.

That is, when iterating through a collection-view of a **LinkedHashMap**, the elements will be returned in the order in which they were inserted. You can also create a **LinkedHashMap** that returns its elements in the order in which they were last accessed. **LinkedHashMap** is a generic class that has this declaration:

```
class LinkedHashMap<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

LinkedHashMap defines the following constructors:

```
LinkedHashMap()
LinkedHashMap(Map<? extends K, ? extends V> m)
LinkedHashMap(int capacity)
LinkedHashMap(int capacity, float fillRatio)
LinkedHashMap(int capacity, float fillRatio, boolean Order)
```

The first form constructs a default **LinkedHashMap**. The second form initializes the **LinkedHashMap** with the elements from *m*. The third form initializes the capacity. The fourth form initializes both capacity and fill ratio. The meaning of capacity and fill ratio are the same as for **HashMap**. The default capacity is 16. The default ratio is 0.75. The last form allows you to specify whether the elements will be stored in the linked list by insertion order, or by order of last access. If *Order* is **true**, then access order is used. If *Order* is **false**, then insertion order is used.

LinkedHashMap adds only one method to those defined by **HashMap**. This method is **removeEldestEntry()**, and it is shown here:

```
protected boolean removeEldestEntry(Map.Entry<K, V> e)
```

This method is called by **put()** and **putAll()**. The oldest entry is passed in *e*. By default, this method returns **false** and does nothing. However, if you override this method, then you can have the **LinkedHashMap** remove the oldest entry in the map. To do this, have your override return **true**. To keep the oldest entry, return **false**.

The IdentityHashMap Class

IdentityHashMap extends **AbstractMap** and implements the **Map** interface. It is similar to **HashMap** except that it uses reference equality when comparing elements. **IdentityHashMap** is a generic class that has this declaration:

```
class IdentityHashMap<K, V>
```

Here, **K** specifies the type of key, and **V** specifies the type of value. The API documentation explicitly states that **IdentityHashMap** is not for general use.

The EnumMap Class

EnumMap extends **AbstractMap** and implements **Map**. It is specifically for use with keys of an **enum** type. It is a generic class that has this declaration:

```
class EnumMap<K extends Enum<K>, V>
```

Here, **K** specifies the type of key, and **V** specifies the type of value. Notice that **K** must extend **Enum<K>**, which enforces the requirement that the keys must be of an **enum** type.

EnumMap defines the following constructors:

```
EnumMap(Class<K> kType)
EnumMap(Map<K, ? extends V> m)
EnumMap(EnumMap<K, ? extends V> em)
```

The first constructor creates an empty **EnumMap** of type *kType*. The second creates an **EnumMap** map that contains the same entries as *m*. The third creates an **EnumMap** initialized with the values in *em*.

EnumMap defines no methods of its own.

Comparators

Both **TreeSet** and **TreeMap** store elements in sorted order. However, it is the comparator that defines precisely what “sorted order” means. By default, these classes store their elements by using what Java refers to as “natural ordering,” which is usually the ordering that you would expect (A before B, 1 before 2, and so forth). If you want to order elements a different way, then specify a **Comparator** when you construct the set or map. Doing so gives you the ability to govern precisely how elements are stored within sorted collections and maps.

Comparator is a generic interface that has this declaration:

```
interface Comparator<T>
```

Here, **T** specifies the type of objects being compared.

Prior to JDK 8, the **Comparator** interface defined only two methods: **compare()** and **equals()**. The **compare()** method, shown here, compares two elements for order:

```
int compare(T obj1, T obj2)
```

obj1 and *obj2* are the objects to be compared. Normally, this method returns zero if the objects are equal. It returns a positive value if *obj1* is greater than *obj2*. Otherwise, a negative value is returned. The method can throw a **ClassCastException** if the types of the objects are not compatible for comparison. By implementing **compare()**, you can alter the way that objects are ordered. For example, to sort in reverse order, you can create a comparator that reverses the outcome of a comparison.

The **equals()** method, shown here, tests whether an object equals the invoking comparator:

```
boolean equals(object obj)
```

Here, *obj* is the object to be tested for equality. The method returns **true** if *obj* and the invoking object are both **Comparator** objects and use the same ordering. Otherwise, it returns **false**. Overriding **equals()** is not necessary, and most simple comparators will not do so.

For many years, the preceding two methods were the only methods defined by **Comparator**. With the release of JDK 8, the situation has dramatically changed. JDK 8 adds significant new functionality to **Comparator** through the use of default and static interface methods. Each is described here.

You can obtain a comparator that reverses the ordering of the comparator on which it is called by using **reversed()**, shown here:

```
default Comparator<T> reversed()
```

It returns the reverse comparator. For example, assuming a comparator that uses natural ordering for the characters A through Z, a reverse order comparator would put B before A, C before B, and so on.

A method related to **reversed()** is **reverseOrder()**, shown next:

```
static <T extends Comparable<? super T>> Comparator<T> reverseOrder()
```

It returns a comparator that reverses the natural order of the elements. Conversely, you can obtain a comparator that uses natural ordering by calling the static method **naturalOrder()**, shown next:

```
static <T extends Comparable<? super T>> Comparator<T> naturalOrder()
```

If you want a comparator that can handle **null** values, use **nullsFirst()** or **nullsLast()**, shown here:

```
static <T> Comparator<T> nullsFirst(Comparator<? super T> comp)
static <T> Comparator<T> nullsLast(Comparator<? super T> comp)
```

The **nullsFirst()** method returns a comparator that views **null** values as less than other values. The **nullsLast()** method returns a comparator that views **null** values as greater than other values. In both cases, if the two values being compared are non-**null**, *comp* performs the comparison. If *comp* is passed **null**, then all non-**null** values are viewed as equivalent.

Another default method added by JDK 8 is **thenComparing()**. It returns a comparator that performs a second comparison when the outcome of the first comparison indicates that the objects being compared are equal. Thus, it can be used to create a “compare by X then compare by Y” sequence. For example, when comparing cities, the first comparison might compare names, with the second comparison comparing states. (Therefore, Springfield, Illinois, would come before Springfield, Missouri, assuming normal, alphabetical order.) The **thenComparing()** method has three forms. The first, shown here, lets you specify the second comparator by passing an instance of **Comparator**:

```
default Comparator<T> thenComparing(Comparator<? super T> thenByComp)
```

Here, *thenByComp* specifies the comparator that is called if the first comparison returns equal.

The next versions of **thenComparing()** let you specify the standard functional interface **Function** (defined by **java.util.function**). They are shown here:

```
default <U extends Comparable<? super U>> Comparator<T>
    thenComparing(Function<? super T, ? extends U> getKey)
```

```
default <U> Comparator<T>
    thenComparing(Function<? super T, ? extends U> getKey,
        Comparator<? super U> keyComp)
```

In both, *getKey* refers to function that obtains the next comparison key, which is used if the first comparison returns equal. In the second version, *keyComp* specifies the comparator used to compare keys. (Here, and in subsequent uses, U specifies the type of the key.)

Comparator also adds the following specialized versions of “then comparing” methods for the primitive types:

```
default Comparator<T>
    thenComparingDouble(ToDoubleFunction<? super T> getKey)
default Comparator<T>
    thenComparingInt(ToIntFunction<? super T> getKey)
default Comparator<T>
    thenComparingLong(ToLongFunction<? super T> getKey)
```

In all methods, *getKey* refers to a function that obtains the next comparison key.

Finally, JDK 8 adds to **Comparator** a method called **comparing()**. It returns a comparator that obtains its comparison key from a function passed to the method. There are two versions of **comparing()**, shown here:

```
static <T, U extends Comparable<? super U>> Comparator<T>
    comparing(Function<? super T, ? extends U> getKey)
static <T, U> Comparator<T>
    comparing(Function<? super T, ? extends U> getKey,
              Comparator<? super U> keyComp)
```

In both, *getKey* refers to a function that obtains the next comparison key. In the second version, *keyComp* specifies the comparator used to compare keys. **Comparator** also adds the following specialized versions of these methods for the primitive types:

```
static <T> Comparator<T>
    ComparingDouble(ToDoubleFunction<? super T> getKey)
static <T> Comparator<T>
    ComparingInt(ToIntFunction<? super T> getKey)
static <T> Comparator<T>
    ComparingLong(ToLongFunction<? super T> getKey)
```

In all methods, *getKey* refers to a function that obtains the next comparison key.

Using a Comparator

The following is an example that demonstrates the power of a custom comparator. It implements the **compare()** method for strings that operates in reverse of normal. Thus, it causes a tree set to be sorted in reverse order.

```
// Use a custom comparator.
import java.util.*;

// A reverse comparator for strings.
class MyComp implements Comparator<String> {
```

```
public int compare(String aStr, String bStr) {  
    // Reverse the comparison.  
    return bStr.compareTo(aStr);  
}  
  
// No need to override equals or the default methods.  
}  
  
class CompDemo {  
    public static void main(String args[]) {  
        // Create a tree set.  
        TreeSet<String> ts = new TreeSet<String>(new MyComp());  
  
        // Add elements to the tree set.  
        ts.add("C");  
        ts.add("A");  
        ts.add("B");  
        ts.add("E");  
        ts.add("F");  
        ts.add("D");  
  
        // Display the elements.  
        for(String element : ts)  
            System.out.print(element + " ");  
  
        System.out.println();  
    }  
}
```

As the following output shows, the tree is now sorted in reverse order:

F E D C B A

Look closely at the **MyComp** class, which implements **Comparator** by implementing **compare()**. (As explained earlier, overriding **equals()** is neither necessary nor common. It is also not necessary to override the default methods added by JDK 8.) Inside **compare()**, the **String** method **compareTo()** compares the two strings. However, **bStr**—not **aStr**—invokes **compareTo()**. This causes the outcome of the comparison to be reversed.

Although the way in which the reverse order comparator is implemented by the preceding program is perfectly adequate, beginning with JDK 8, there is another way to approach a solution. It is now possible to simply call **reversed()** on a natural-order comparator. It will return an equivalent comparator, except that it runs in reverse. For example, assuming the preceding program, you can rewrite **MyComp** as a natural-order comparator, as shown here:

```
class MyComp implements Comparator<String> {  
    public int compare(String aStr, String bStr) {  
        return aStr.compareTo(bStr);  
    }  
}
```

Next, you can use the following sequence to create a **TreeSet** that orders its string elements in reverse:

```
MyComp mc = new MyComp(); // Create a comparator
// Pass a reverse order version of MyComp to TreeSet.
TreeSet<String> ts = new TreeSet<String>(mc.reversed());
```

If you plug this new code into the preceding program, it will produce the same results as before. In this case, there is no advantage gained by using **reversed()**. However, in cases in which you need to create both a natural-order comparator and a reversed comparator, then using **reversed()** gives you an easy way to obtain the reverse-order comparator without having to code it explicitly.

Beginning with JDK 8, it is not actually necessary to create the **MyComp** class in the preceding examples because a lambda expression can be easily used instead. For example, you can remove the **MyComp** class entirely and create the string comparator by using this statement:

```
// Use a lambda expression to implement Comparator<String>.
Comparator<String> mc = (aStr, bStr) -> aStr.compareTo(bStr);
```

One other point: in this simple example, it would also be possible to specify a reverse comparator via a lambda expression directly in the call to the **TreeSet()** constructor, as shown here:

```
// Pass a reversed comparator to TreeSet() via a
// lambda expression.
TreeSet<String> ts = new TreeSet<String>(
    (aStr, bStr) -> bStr.compareTo(aStr));
```

By making these changes, the program is substantially shortened, as its final version shown here illustrates:

```
// Use a lambda expression to create a reverse comparator.
import java.util.*;

class CompDemo2 {
    public static void main(String args[]) {

        // Pass a reverse comparator to TreeSet() via a
        // lambda expression.
        TreeSet<String> ts = new TreeSet<String>(
            (aStr, bStr) -> bStr.compareTo(aStr));

        // Add elements to the tree set.
        ts.add("C");
        ts.add("A");
        ts.add("B");
        ts.add("E");
        ts.add("F");
        ts.add("D");
```

```
// Display the elements.  
for(String element : ts)  
    System.out.print(element + " ");  
  
System.out.println();  
}  
}
```

For a more practical example that uses a custom comparator, the following program is an updated version of the **TreeMap** program shown earlier that stores account balances. In the previous version, the accounts were sorted by name, but the sorting began with the first name. The following program sorts the accounts by last name. To do so, it uses a comparator that compares the last name of each account. This results in the map being sorted by last name.

```
// Use a comparator to sort accounts by last name.  
import java.util.*;  
  
// Compare last whole words in two strings.  
class TComp implements Comparator<String> {  
    public int compare(String aStr, String bStr) {  
        int i, j, k;  
  
        // Find index of beginning of last name.  
        i = aStr.lastIndexOf(' ');  
        j = bStr.lastIndexOf(' ');  
  
        k = aStr.substring(i).compareToIgnoreCase(bStr.substring(j));  
        if(k==0) // last names match, check entire name  
            return aStr.compareToIgnoreCase(bStr);  
        else  
            return k;  
    }  
  
    // No need to override equals.  
}  
  
class TreeMapDemo2 {  
    public static void main(String args[]) {  
        // Create a tree map.  
        TreeMap<String, Double> tm = new TreeMap<String, Double>(new TComp());  
  
        // Put elements to the map.  
        tm.put("John Doe", new Double(3434.34));  
        tm.put("Tom Smith", new Double(123.22));  
        tm.put("Jane Baker", new Double(1378.00));  
        tm.put("Tod Hall", new Double(99.22));  
        tm.put("Ralph Smith", new Double(-19.08));  
  
        // Get a set of the entries.  
        Set<Map.Entry<String, Double>> set = tm.entrySet();
```

```

// Display the elements.
for(Map.Entry<String, Double> me : set) {
    System.out.print(me.getKey() + ": ");
    System.out.println(me.getValue());
}
System.out.println();

// Deposit 1000 into John Doe's account.
double balance = tm.get("John Doe");
tm.put("John Doe", balance + 1000);

System.out.println("John Doe's new balance: " +
    tm.get("John Doe"));
}
}

```

Here is the output; notice that the accounts are now sorted by last name:

```

Jane Baker: 1378.0
John Doe: 3434.34
Todd Hall: 99.22
Ralph Smith: -19.08
Tom Smith: 123.22

John Doe's new balance: 4434.34

```

The comparator class **TComp** compares two strings that hold first and last names. It does so by first comparing last names. To do this, it finds the index of the last space in each string and then compares the substrings of each element that begin at that point. In cases where last names are equivalent, the first names are then compared. This yields a tree map that is sorted by last name, and within last name by first name. You can see this because Ralph Smith comes before Tom Smith in the output.

If you are using JDK 8 or later, then there is another way that you could code the preceding program so the map is sorted by last name and then by first name. This approach uses the **thenComparing()** method. Recall that **thenComparing()** lets you specify a second comparator that will be used if the invoking comparator returns equal. This approach is put into action by the following program, which reworks the preceding example to use **thenComparing()**:

```

// Use thenComparing() to sort by last, then first name.
import java.util.*;

// A comparator that compares last names.
class CompLastNames implements Comparator<String> {
    public int compare(String aStr, String bStr) {
        int i, j;

        // Find index of beginning of last name.
        i = aStr.lastIndexOf(' ');
        j = bStr.lastIndexOf(' ');

```

```
        return aStr.substring(i).compareToIgnoreCase(bStr.substring(j));
    }
}

// Sort by entire name when last names are equal.
class CompThenByFirstName implements Comparator<String> {
    public int compare(String aStr, String bStr) {
        int i, j;

        return aStr.compareToIgnoreCase(bStr);
    }
}

class TreeMapDemo2A {
    public static void main(String args[]) {
        // Use thenComparing() to create a comparator that compares
        // last names, then compares entire name when last names match.
        CompLastNames compLN = new CompLastNames();
        Comparator<String> compLastThenFirst =
            compLN.thenComparing(new CompThenByFirstName());

        // Create a tree map.
        TreeMap<String, Double> tm =
            new TreeMap<String, Double>(compLastThenFirst);

        // Put elements to the map.
        tm.put("John Doe", new Double(3434.34));
        tm.put("Tom Smith", new Double(123.22));
        tm.put("Jane Baker", new Double(1378.00));
        tm.put("Tod Hall", new Double(99.22));
        tm.put("Ralph Smith", new Double(-19.08));

        // Get a set of the entries.
        Set<Map.Entry<String, Double>> set = tm.entrySet();

        // Display the elements.
        for(Map.Entry<String, Double> me : set) {
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());
        }
        System.out.println();

        // Deposit 1000 into John Doe's account.
        double balance = tm.get("John Doe");
        tm.put("John Doe", balance + 1000);

        System.out.println("John Doe's new balance: " +
            tm.get("John Doe"));
    }
}
```

This version produces the same output as before. It differs only in how it accomplishes its job. To begin, notice that a comparator called **CompLastNames** is created. This comparator

compares only the last names. A second comparator, called **CompThenByFirstName**, compares the entire name, starting with the first name. Next, the **TreeMap** is then created by the following sequence:

```
CompLastNames compLN = new CompLastNames();
Comparator<String> compLastThenFirst =
    compLN.thenComparing(new CompThenByFirstName());
```

Here, the primary comparator is **compLN**. It is an instance of **CompLastNames**. On it is called **thenComparing()**, passing in an instance of **CompThenByFirstName**. The result is assigned to the comparator called **compLastThenFirst**. This comparator is used to construct the **TreeMap**, as shown here:

```
TreeMap<String, Double> tm =
    new TreeMap<String, Double>(compLastThenFirst);
```

Now, whenever the last names of the items being compared are equal, the entire name, beginning with the first name, is used to order the two. This means that names are ordered based on last name, and within last names, by first names.

One last point: in the interest of clarity, this example explicitly creates two comparator classes called **CompLastNames** and **ThenByFirstNames**, but lambda expressions could have been used instead. You might want to try this on your own. Just follow the same general approach described for the **CompDemo2** example shown earlier.

The Collection Algorithms

The Collections Framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the **Collections** class. They are summarized in Table 18-15. As explained earlier, beginning with JDK 5 all of the algorithms were retrofitted for generics.

Several of the methods can throw a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or an **UnsupportedOperationException**, which occurs when an attempt is made to modify an unmodifiable collection. Other exceptions are possible, depending on the method.

One thing to pay special attention to is the set of **checked** methods, such as **checkedCollection()**, which returns what the API documentation refers to as a “dynamically typesafe view” of a collection. This view is a reference to the collection that monitors insertions into the collection for type compatibility at run time. An attempt to insert an incompatible element will cause a **ClassCastException**. Using such a view is especially helpful during debugging because it ensures that the collection always contains valid elements. Related methods include **checkedSet()**, **checkedList()**, **checkedMap()**, and so on. They obtain a type-safe view for the indicated collection.

Notice that several methods, such as **synchronizedList()** and **synchronizedSet()**, are used to obtain synchronized (*thread-safe*) copies of the various collections. As a general rule, the standard collections implementations are not synchronized. You must use the synchronization algorithms to provide synchronization. One other point: iterators to synchronized collections must be used within **synchronized** blocks.

Method	Description
static <T> boolean addAll(Collection <? super T> c, T... elements)	Inserts the elements specified by <i>elements</i> into the collection specified by <i>c</i> . Returns true if the elements were added and false otherwise.
static <T> Queue<T> asLifoQueue(Deque<T> c)	Returns a last-in, first-out view of <i>c</i> .
static <T> int binarySearch(List<? extends T> list, T value, Comparator<? super T> c)	Searches for <i>value</i> in <i>list</i> ordered according to <i>c</i> . Returns the position of <i>value</i> in <i>list</i> , or a negative value if <i>value</i> is not found.
static <T> int binarySearch(List<? extends Comparable<? super T>> list, T value)	Searches for <i>value</i> in <i>list</i> . The list must be sorted. Returns the position of <i>value</i> in <i>list</i> , or a negative value if <i>value</i> is not found.
static <E> Collection<E> checkedCollection(Collection<E> c, Class<E> t)	Returns a run-time type-safe view of a collection. An attempt to insert an incompatible element will cause a ClassCastException .
static <E> List<E> checkedList(List<E> c, Class<E> t)	Returns a run-time type-safe view of a List . An attempt to insert an incompatible element will cause a ClassCastException .
static <K, V> Map<K, V> checkedMap(Map<K, V> c, Class<K> keyT, Class<V> valueT)	Returns a run-time type-safe view of a Map . An attempt to insert an incompatible element will cause a ClassCastException .
static <K, V> NavigableMap<K, V> checkedNavigableMap(NavigableMap<K, V> nm, Class<E> keyT, Class<V> valueT)	Returns a run-time type-safe view of a NavigableMap . An attempt to insert an incompatible element will cause a ClassCastException . (Added by JDK 8.)
static <E> NavigableSet<E> checkedNavigableSet(NavigableSet<E> ns, Class<E> t)	Returns a run-time type-safe view of a NavigableSet . An attempt to insert an incompatible element will cause a ClassCastException . (Added by JDK 8.)
static <E> Queue<E> checkedQueue(Queue<E> q, Class<E> t)	Returns a run-time type-safe view of a Queue . An attempt to insert an incompatible element will cause a ClassCastException . (Added by JDK 8.)
static <E> List<E> checkedSet(Set<E> c, Class<E> t)	Returns a run-time type-safe view of a Set . An attempt to insert an incompatible element will cause a ClassCastException .

Table 18-15 The Algorithms Defined by **Collections**

Method	Description
static <K, V> SortedMap<K, V> checkedSortedMap(SortedMap<K, V> <i>c</i> , Class<K> <i>keyT</i> , Class<V> <i>valueT</i>)	Returns a run-time type-safe view of a SortedMap . An attempt to insert an incompatible element will cause a ClassCastException .
static <E> SortedSet<E> checkedSortedSet(SortedSet<E> <i>c</i> , Class<E> <i>t</i>)	Returns a run-time type-safe view of a SortedSet . An attempt to insert an incompatible element will cause a ClassCastException .
static <T> void copy(List<? super T> <i>list1</i> , List<? extends T> <i>list2</i>)	Copies the elements of <i>list2</i> to <i>list1</i> .
static boolean disjoint(Collection<?> <i>a</i> , Collection<?> <i>b</i>)	Compares the elements in <i>a</i> to elements in <i>b</i> . Returns true if the two collections contain no common elements (i.e., the collections contain disjoint sets of elements). Otherwise, returns false .
static <T> Enumeration<T> emptyEnumeration()	Returns an empty enumeration, which is an enumeration with no elements.
static <T> Iterator<T> emptyIterator()	Returns an empty iterator, which is an iterator with no elements.
static <T> List<T> emptyList()	Returns an immutable, empty List object of the inferred type.
static <T> ListIterator<T> emptyListIterator()	Returns an empty list iterator, which is a list iterator that has no elements.
static <K, V> Map<K, V> emptyMap()	Returns an immutable, empty Map object of the inferred type.
static <K, V> NavigableMap<K, V> emptyNavigableMap()	Returns an immutable, empty NavigableMap object of the inferred type. (Added by JDK 8.)
static <E> NavigableSet<E> emptyNavigableSet()	Returns an immutable, empty NavigableSet object of the inferred type. (Added by JDK 8.)
static <T> Set<T> emptySet()	Returns an immutable, empty Set object of the inferred type.
static <K, V> SortedMap<K, V> emptySortedMap()	Returns an immutable, empty SortedMap object of the inferred type. (Added by JDK 8.)
static <E> SortedSet<E> emptySortedSet()	Returns an immutable, empty SortedSet object of the inferred type. (Added by JDK 8.)
static <T> Enumeration<T> enumeration(Collection<T> <i>c</i>)	Returns an enumeration over <i>c</i> . (See “The Enumeration Interface,” later in this chapter.)
static <T> void fill(List<? super T> <i>list</i> , T <i>obj</i>)	Assigns <i>obj</i> to each element of <i>list</i> .
static int frequency(Collection<?> <i>c</i> , object <i>obj</i>)	Counts the number of occurrences of <i>obj</i> in <i>c</i> and returns the result.

Table 18-15 The Algorithms Defined by **Collections** (*continued*)

Method	Description
static int indexOfSubList(List<?> list, List<?> subList)	Searches <i>list</i> for the first occurrence of <i>subList</i> . Returns the index of the first match, or -1 if no match is found.
static int lastIndexOfSubList(List<?> list, List<?> subList)	Searches <i>list</i> for the last occurrence of <i>subList</i> . Returns the index of the last match, or -1 if no match is found.
static <T> ArrayList<T> list(Enumeration<T> enum)	Returns an ArrayList that contains the elements of <i>enum</i> .
static <T> T max(Collection<? extends T> c, Comparator<? super T> comp)	Returns the maximum element in <i>c</i> as determined by <i>comp</i> .
static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> c)	Returns the maximum element in <i>c</i> as determined by natural ordering. The collection need not be sorted.
static <T> T min(Collection<? extends T> c, Comparator<? super T> comp)	Returns the minimum element in <i>c</i> as determined by <i>comp</i> . The collection need not be sorted.
static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> c)	Returns the minimum element in <i>c</i> as determined by natural ordering.
static <T> List<T> nCopies(int num, T obj)	Returns <i>num</i> copies of <i>obj</i> contained in an immutable list. <i>num</i> must be greater than or equal to zero.
static <E> Set<E> newSetFromMap(Map<E, Boolean> m)	Creates and returns a set backed by the map specified by <i>m</i> , which must be empty at the time this method is called.
static <T> boolean replaceAll(List<T> list, T old, T new)	Replaces all occurrences of <i>old</i> with <i>new</i> in <i>list</i> . Returns true if at least one replacement occurred. Returns false otherwise.
static void reverse(List<T> list)	Reverses the sequence in <i>list</i> .
static <T> Comparator<T> reverseOrder(Comparator<T> comp)	Returns a reverse comparator based on the one passed in <i>comp</i> . That is, the returned comparator reverses the outcome of a comparison that uses <i>comp</i> .
static <T> Comparator<T> reverseOrder()	Returns a reverse comparator, which is a comparator that reverses the outcome of a comparison between two elements.
static void rotate(List<T> list, int n)	Rotates <i>list</i> by <i>n</i> places to the right. To rotate left, use a negative value for <i>n</i> .
static void shuffle(List<T> list, Random r)	Shuffles (i.e., randomizes) the elements in <i>list</i> by using <i>r</i> as a source of random numbers.
static void shuffle(List<T> list)	Shuffles (i.e., randomizes) the elements in <i>list</i> .

Table 18-15 The Algorithms Defined by **Collections** (*continued*)

Method	Description
static <T> Set<T> singleton(T <i>obj</i>)	Returns <i>obj</i> as an immutable set. This is an easy way to convert a single object into a set.
static <T> List<T> singletonList(T <i>obj</i>)	Returns <i>obj</i> as an immutable list. This is an easy way to convert a single object into a list.
static <K, V> Map<K, V> singletonMap(K <i>k</i> , V <i>v</i>)	Returns the key/value pair <i>k/v</i> as an immutable map. This is an easy way to convert a single key/value pair into a map.
static <T> void sort(List<T> <i>list</i> , Comparator<? super T> <i>comp</i>)	Sorts the elements of <i>list</i> as determined by <i>comp</i> .
static <T extends Comparable<? super T>> void sort(List<T> <i>list</i>)	Sorts the elements of <i>list</i> as determined by their natural ordering.
static void swap(List<?> <i>list</i> , int <i>idx1</i> , int <i>idx2</i>)	Exchanges the elements in <i>list</i> at the indices specified by <i>idx1</i> and <i>idx2</i> .
static <T> Collection<T> synchronizedCollection(Collection<T> <i>c</i>)	Returns a thread-safe collection backed by <i>c</i> .
static <T> List<T> synchronizedList(List<T> <i>list</i>)	Returns a thread-safe list backed by <i>list</i> .
static <K, V> Map<K, V> synchronizedMap(Map<K, V> <i>m</i>)	Returns a thread-safe map backed by <i>m</i> .
static <K, V> NavigableMap<K, V> synchronizedNavigableMap(NavigableMap<K, V> <i>nm</i>)	Returns a synchronized navigable map backed by <i>nm</i> . (Added by JDK 8.)
static <T> NavigableSet<T> synchronizedNavigableSet(NavigableSet<T> <i>ns</i>)	Returns a synchronized navigable set backed by <i>ns</i> . (Added by JDK 8.)
static <T> Set<T> synchronizedSet(Set<T> <i>s</i>)	Returns a thread-safe set backed by <i>s</i> .
static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> <i>sm</i>)	Returns a thread-safe sorted map backed by <i>sm</i> .
static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> <i>ss</i>)	Returns a thread-safe sorted set backed by <i>ss</i> .
static <T> Collection<T> unmodifiableCollection(Collection<? extends T> <i>c</i>)	Returns an unmodifiable collection backed by <i>c</i> .
static <T> List<T> unmodifiableList(List<? extends T> <i>list</i>)	Returns an unmodifiable list backed by <i>list</i> .
static <K, V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> <i>m</i>)	Returns an unmodifiable map backed by <i>m</i> .
static <K, V> NavigableMap<K, V> unmodifiableNavigableMap(NavigableMap<K, ? extends V> <i>nm</i>)	Returns an unmodifiable navigable map backed by <i>nm</i> . (Added by JDK 8.)

Table 18-15 The Algorithms Defined by **Collections** (*continued*)

Method	Description
static <T> NavigableSet<T> unmodifiableNavigableSet(NavigableSet<T> ns)	Returns an unmodifiable navigable set backed by <i>ns</i> . (Added by JDK 8.)
static <T> Set<T> unmodifiableSet(Set<? extends T> s)	Returns an unmodifiable set backed by <i>s</i> .
static <K, V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? extends V> sm)	Returns an unmodifiable sorted map backed by <i>sm</i> .
static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> ss)	Returns an unmodifiable sorted set backed by <i>ss</i> .

Table 18-15 The Algorithms Defined by **Collections** (*continued*)

The set of methods that begins with **unmodifiable** returns views of the various collections that cannot be modified. These will be useful when you want to grant some process read—but not write—capabilities on a collection.

Collections defines three static variables: **EMPTY_SET**, **EMPTY_LIST**, and **EMPTY_MAP**. All are immutable.

The following program demonstrates some of the algorithms. It creates and initializes a linked list. The **reverseOrder()** method returns a **Comparator** that reverses the comparison of **Integer** objects. The list elements are sorted according to this comparator and then are displayed. Next, the list is randomized by calling **shuffle()**, and then its minimum and maximum values are displayed.

```
// Demonstrate various algorithms.
import java.util.*;

class AlgorithmsDemo {
    public static void main(String args[]) {

        // Create and initialize linked list.
        LinkedList<Integer> ll = new LinkedList<Integer>();
        ll.add(-8);
        ll.add(20);
        ll.add(-20);
        ll.add(8);

        // Create a reverse order comparator.
        Comparator<Integer> r = Collections.reverseOrder();

        // Sort list by using the comparator.
        Collections.sort(ll, r);

        System.out.print("List sorted in reverse: ");
        for(int i : ll)
            System.out.print(i+ " ");
    }
}
```

```

        System.out.println();

        // Shuffle list.
        Collections.shuffle(l1);

        // Display randomized list.
        System.out.print("List shuffled: ");
        for(int i : l1)
            System.out.print(i + " ");

        System.out.println();
        System.out.println("Minimum: " + Collections.min(l1));
        System.out.println("Maximum: " + Collections.max(l1));
    }
}

```

Output from this program is shown here:

```

List sorted in reverse: 20 8 -8 -20
List shuffled: 20 -20 8 -8
Minimum: -20
Maximum: 20

```

Notice that **min()** and **max()** operate on the list after it has been shuffled. Neither requires a sorted list for its operation.

Arrays

The **Arrays** class provides various methods that are useful when working with arrays. These methods help bridge the gap between collections and arrays. Each method defined by **Arrays** is examined in this section.

The **asList()** method returns a **List** that is backed by a specified array. In other words, both the list and the array refer to the same location. It has the following signature:

```
static <T> List<T> asList(T... array)
```

Here, *array* is the array that contains the data.

The **binarySearch()** method uses a binary search to find a specified value. This method must be applied to sorted arrays. Here are some of its forms. (Additional forms let you search a subrange):

```

static int binarySearch(byte array[ ], byte value)
static int binarySearch(char array[ ], char value)
static int binarySearch(double array[ ], double value)
static int binarySearch(float array[ ], float value)
static int binarySearch(int array[ ], int value)
static int binarySearch(long array[ ], long value)
static int binarySearch(short array[ ], short value)
static int binarySearch(Object array[ ], Object value)
static <T> int binarySearch(T[ ] array, T value, Comparator<? super T> c)

```

Here, *array* is the array to be searched, and *value* is the value to be located. The last two forms throw a **ClassCastException** if *array* contains elements that cannot be compared (for example, **Double** and **StringBuffer**) or if *value* is not compatible with the types in *array*. In the last form, the **Comparator** *c* is used to determine the order of the elements in *array*. In all cases, if *value* exists in *array*, the index of the element is returned. Otherwise, a negative value is returned.

The **copyOf()** method returns a copy of an array and has the following forms:

```
static boolean[ ] copyOf(boolean[ ] source, int len)
static byte[ ] copyOf(byte[ ] source, int len)
static char[ ] copyOf(char[ ] source, int len)
static double[ ] copyOf(double[ ] source, int len)
static float[ ] copyOf(float[ ] source, int len)
static int[ ] copyOf(int[ ] source, int len)
static long[ ] copyOf(long[ ] source, int len)
static short[ ] copyOf(short[ ] source, int len)
static <T> T[ ] copyOf(T[ ] source, int len)
static <T,U> T[ ] copyOf(U[ ] source, int len, Class<? extends T[ ]> resultT)
```

The original array is specified by *source*, and the length of the copy is specified by *len*. If the copy is longer than *source*, then the copy is padded with zeros (for numeric arrays), **nulls** (for object arrays), or **false** (for boolean arrays). If the copy is shorter than *source*, then the copy is truncated. In the last form, the type of *resultT* becomes the type of the array returned. If *len* is negative, a **NegativeArraySizeException** is thrown. If *source* is **null**, a **NullPointerException** is thrown. If *resultT* is incompatible with the type of *source*, an **ArrayStoreException** is thrown.

The **copyOfRange()** method returns a copy of a range within an array and has the following forms:

```
static boolean[ ] copyOfRange(boolean[ ] source, int start, int end)
static byte[ ] copyOfRange(byte[ ] source, int start, int end)
static char[ ] copyOfRange(char[ ] source, int start, int end)
static double[ ] copyOfRange(double[ ] source, int start, int end)
static float[ ] copyOfRange(float[ ] source, int start, int end)
static int[ ] copyOfRange(int[ ] source, int start, int end)
static long[ ] copyOfRange(long[ ] source, int start, int end)
static short[ ] copyOfRange(short[ ] source, int start, int end)
static <T> T[ ] copyOfRange(T[ ] source, int start, int end)
static <T,U> T[ ] copyOfRange(U[ ] source, int start, int end,
                           Class<? extends T[ ]> resultT)
```

The original array is specified by *source*. The range to copy is specified by the indices passed via *start* and *end*. The range runs from *start* to *end* – 1. If the range is longer than *source*, then the copy is padded with zeros (for numeric arrays), **nulls** (for object arrays), or **false** (for boolean arrays). In the last form, the type of *resultT* becomes the type of the array returned. If *start* is negative or greater than the length of *source*, an **ArrayIndexOutOfBoundsException** is thrown. If *start* is greater than *end*, an

IllegalArgumentException is thrown. If *source* is **null**, a **NullPointerException** is thrown. If *resultT* is incompatible with the type of *source*, an **ArrayStoreException** is thrown.

The **equals()** method returns **true** if two arrays are equivalent. Otherwise, it returns **false**. The **equals()** method has the following forms:

```
static boolean equals(boolean array1[ ], boolean array2[ ])
static boolean equals(byte array1[ ], byte array2[ ])
static boolean equals(char array1[ ], char array2[ ])
static boolean equals(double array1[ ], double array2[ ])
static boolean equals(float array1[ ], float array2[ ])
static boolean equals(int array1[ ], int array2[ ])
static boolean equals(long array1[ ], long array2[ ])
static boolean equals(short array1[ ], short array2[ ])
static boolean equals(Object array1[ ], Object array2[ ])
```

Here, *array1* and *array2* are the two arrays that are compared for equality.

The **deepEquals()** method can be used to determine if two arrays, which might contain nested arrays, are equal. It has this declaration:

```
static boolean deepEquals(Object[ ] a, Object[ ] b)
```

It returns **true** if the arrays passed in *a* and *b* contain the same elements. If *a* and *b* contain nested arrays, then the contents of those nested arrays are also checked. It returns **false** if the arrays, or any nested arrays, differ.

The **fill()** method assigns a value to all elements in an array. In other words, it fills an array with a specified value. The **fill()** method has two versions. The first version, which has the following forms, fills an entire array:

```
static void fill(boolean array[ ], boolean value)
static void fill(byte array[ ], byte value)
static void fill(char array[ ], char value)
static void fill(double array[ ], double value)
static void fill(float array[ ], float value)
static void fill(int array[ ], int value)
static void fill(long array[ ], long value)
static void fill(short array[ ], short value)
static void fill(Object array[ ], Object value)
```

Here, *value* is assigned to all elements in *array*. The second version of the **fill()** method assigns a value to a subset of an array.

The **sort()** method sorts an array so that it is arranged in ascending order. The **sort()** method has two versions. The first version, shown here, sorts the entire array:

```
static void sort(byte array[ ])
static void sort(char array[ ])
static void sort(double array[ ])
static void sort(float array[ ])
static void sort(int array[ ])
static void sort(long array[ ]) 
```

```
static void sort(short array[ ])
static void sort(Object array[ ])
static <T> void sort(T array[ ], Comparator<? super T> c)
```

Here, *array* is the array to be sorted. In the last form, *c* is a **Comparator** that is used to order the elements of *array*. The last two forms can throw a **ClassCastException** if elements of the array being sorted are not comparable. The second version of **sort()** enables you to specify a range within an array that you want to sort.

JDK 8 adds several new methods to **Arrays**. Perhaps the most important is **parallelSort()** because it sorts, into ascending order, portions of an array in parallel and then merges the results. This approach can greatly speed up sorting times. Like **sort()**, there are two basic types of **parallelSort()**, each with several overloads. The first type sorts the entire array. It is shown here:

```
static void parallelSort(byte array[ ])
static void parallelSort(char array[ ])
static void parallelSort(double array[ ])
static void parallelSort(float array[ ])
static void parallelSort(int array[ ])
static void parallelSort(long array[ ])
static void parallelSort(short array[ ])
static <T extends Comparable<? super T>> void parallelSort(T array[ ])
static <T> void parallelSort(T array[ ], Comparator<? super T> c)
```

Here, *array* is the array to be sorted. In the last form, *c* is a comparator that is used to order the elements in the array. The last two forms can throw a **ClassCastException** if the elements of the array being sorted are not comparable. The second version of **parallelSort()** enables you to specify a range within the array that you want to sort.

JDK 8 gives **Arrays** support for spliterators by including the **spliterator()** method. It has two basic forms. The first type returns a spliterator to an entire array. It is shown here:

```
static Spliterator.OfDouble spliterator(double array[ ])
static Spliterator.OfInt spliterator(int array[ ])
static Spliterator.OfLong spliterator(long array[ ])
static <T> Spliterator spliterator(T array[ ])
```

Here, *array* is the array that the spliterator will cycle through. The second version of **spliterator()** enables you to specify a range to iterate within the array.

Beginning with JDK 8, **Arrays** supports the new **Stream** interface (see Chapter 29) by including the **stream()** method. It has two forms. The first is shown here:

```
static DoubleStream stream(double array[ ])
static IntStream stream(int array[ ])
static LongStream stream(long array[ ])
static <T> Stream stream(T array[ ])
```

Here, *array* is the array to which the stream will refer. The second version of **stream()** enables you to specify a range within the array.

In addition to those just discussed, JDK 8 adds three other new methods. Two are related: `setAll()` and `parallelSetAll()`. Both assign values to all of the elements, but `parallelSetAll()` works in parallel. Here is an example of each:

```
static void setAll(double array[ ],
                  IntToDoubleFunction<? extends T> genVal)
static void parallelSetAll(double array[ ],
                           IntToDoubleFunction<? extends T> genVal)
```

Several overloads exist for each of these that handle types `int`, `long`, and generic.

Finally, JDK 8 includes one of the more intriguing additions to **Arrays**. It is called `parallelPrefix()`, and it modifies an array so that each element contains the cumulative result of an operation applied to all previous elements. For example, if the operation is multiplication, then on return, the array elements will contain the values associated with the running product of the original values. It has several overloads. Here is one example:

```
static void parallelPrefix(double array[ ], DoubleBinaryOperator func)
```

Here, `array` is the array being acted upon, and `func` specifies the operation applied. (**DoubleBinaryOperator** is a functional interface defined in `java.util.function`.) Many other versions are provided, including those that operate on types `int`, `long`, and generic, and those that let you specify a range within the array on which to operate.

Arrays also provides `toString()` and `hashCode()` for the various types of arrays. In addition, `deepToString()` and `deepHashCode()` are provided, which operate effectively on arrays that contain nested arrays.

The following program illustrates how to use some of the methods of the **Arrays** class:

```
// Demonstrate Arrays
import java.util.*;

class ArraysDemo {
    public static void main(String args[]) {

        // Allocate and initialize array.
        int array[] = new int[10];
        for(int i = 0; i < 10; i++)
            array[i] = -3 * i;

        // Display, sort, and display the array.
        System.out.print("Original contents: ");
        display(array);
        Arrays.sort(array);
        System.out.print("Sorted: ");
        display(array);

        // Fill and display the array.
        Arrays.fill(array, 2, 6, -1);
        System.out.print("After fill(): ");
        display(array);

        // Sort and display the array.
    }
}
```

```
Arrays.sort(array);
System.out.print("After sorting again: ");
display(array);

// Binary search for -9.
System.out.print("The value -9 is at location ");
int index =
    Arrays.binarySearch(array, -9);

System.out.println(index);
}

static void display(int array[]) {
    for(int i: array)
        System.out.print(i + " ");

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

The following is the output from this program:

```
Original contents: 0 -3 -6 -9 -12 -15 -18 -21 -24 -27
Sorted: -27 -24 -21 -18 -15 -12 -9 -6 -3 0
After fill(): -27 -24 -1 -1 -1 -9 -6 -3 0
After sorting again: -27 -24 -9 -6 -3 -1 -1 -1 -1 0
The value -9 is at location 2
```

The Legacy Classes and Interfaces

As explained at the start of this chapter, early versions of **java.util** did not include the Collections Framework. Instead, it defined several classes and an interface that provided an ad hoc method of storing objects. When collections were added (by J2SE 1.2), several of the original classes were reengineered to support the collection interfaces. Thus, they are now technically part of the Collections Framework. However, where a modern collection duplicates the functionality of a legacy class, you will usually want to use the newer collection class. In general, the legacy classes are supported because there is still code that uses them.

One other point: none of the modern collection classes described in this chapter are synchronized, but all the legacy classes are synchronized. This distinction may be important in some situations. Of course, you can easily synchronize collections by using one of the algorithms provided by **Collections**.

The legacy classes defined by **java.util** are shown here:

Dictionary	Hashtable	Properties	Stack	Vector
------------	-----------	------------	-------	--------

There is one legacy interface called **Enumeration**. The following sections examine **Enumeration** and each of the legacy classes, in turn.

The Enumeration Interface

The **Enumeration** interface defines the methods by which you can *enumerate* (obtain one at a time) the elements in a collection of objects. This legacy interface has been superseded by **Iterator**. Although not deprecated, **Enumeration** is considered obsolete for new code. However, it is used by several methods defined by the legacy classes (such as **Vector** and **Properties**) and is used by several other API classes. Because it is still in use, it was retrofitted for generics by JDK 5. It has this declaration:

```
interface Enumeration<E>
```

where **E** specifies the type of element being enumerated.

Enumeration specifies the following two methods:

```
boolean hasMoreElements( )
E nextElement( )
```

When implemented, **hasMoreElements()** must return **true** while there are still more elements to extract, and **false** when all the elements have been enumerated. **nextElement()** returns the next object in the enumeration. That is, each call to **nextElement()** obtains the next object in the enumeration. It throws **NoSuchElementException** when the enumeration is complete.

Vector

Vector implements a dynamic array. It is similar to **ArrayList**, but with two differences: **Vector** is synchronized, and it contains many legacy methods that duplicate the functionality of methods defined by the Collections Framework. With the advent of collections, **Vector** was reengineered to extend **AbstractList** and to implement the **List** interface. With the release of JDK 5, it was retrofitted for generics and reengineered to implement **Iterable**. This means that **Vector** is fully compatible with collections, and a **Vector** can have its contents iterated by the enhanced **for** loop.

Vector is declared like this:

```
class Vector<E>
```

Here, **E** specifies the type of element that will be stored.

Here are the **Vector** constructors:

```
Vector( )
Vector(int size)
Vector(int size, int incr)
Vector(Collection<? extends E> c)
```

The first form creates a default vector, which has an initial size of 10. The second form creates a vector whose initial capacity is specified by *size*. The third form creates a vector whose initial capacity is specified by *size* and whose increment is specified by *incr*. The increment specifies the number of elements to allocate each time that a vector is resized upward. The fourth form creates a vector that contains the elements of collection *c*.

All vectors start with an initial capacity. After this initial capacity is reached, the next time that you attempt to store an object in the vector, the vector automatically allocates space for that object plus extra room for additional objects. By allocating more than just the required memory, the vector reduces the number of allocations that must take place as the vector grows. This reduction is important, because allocations are costly in terms of time. The amount of extra space allocated during each reallocation is determined by the increment that you specify when you create the vector. If you don't specify an increment, the vector's size is doubled by each allocation cycle.

Vector defines these protected data members:

```
int capacityIncrement;
int elementCount;
Object[ ] elementData;
```

The increment value is stored in **capacityIncrement**. The number of elements currently in the vector is stored in **elementCount**. The array that holds the vector is stored in **elementData**.

In addition to the collections methods specified by **List**, **Vector** defines several legacy methods, which are summarized in Table 18-16.

Because **Vector** implements **List**, you can use a vector just like you use an **ArrayList** instance. You can also manipulate one using its legacy methods. For example, after you instantiate a **Vector**, you can add an element to it by calling **addElement()**. To obtain the element at a specific location, call **elementAt()**. To obtain the first element in the vector, call **firstElement()**. To retrieve the last element, call **lastElement()**. You can obtain the index of an element by using **indexOf()** and **lastIndexOf()**. To remove an element, call **removeElement()** or **removeElementAt()**.

Method	Description
void addElement(E element)	The object specified by <i>element</i> is added to the vector.
int capacity()	Returns the capacity of the vector.
Object clone()	Returns a duplicate of the invoking vector.
boolean contains(Object element)	Returns true if <i>element</i> is contained by the vector, and returns false if it is not.
void copyInto(Object array[])	The elements contained in the invoking vector are copied into the array specified by <i>array</i> .
E elementAt(int index)	Returns the element at the location specified by <i>index</i> .
Enumeration<E> elements()	Returns an enumeration of the elements in the vector.
void ensureCapacity(int size)	Sets the minimum capacity of the vector to <i>size</i> .
E firstElement()	Returns the first element in the vector.
int indexOf(Object element)	Returns the index of the first occurrence of <i>element</i> . If the object is not in the vector, -1 is returned.

Table 18-16 The Legacy Methods Defined by **Vector**

Method	Description
int indexOf(Object <i>element</i> , int <i>start</i>)	Returns the index of the first occurrence of <i>element</i> at or after <i>start</i> . If the object is not in that portion of the vector, -1 is returned.
void insertElementAt(E <i>element</i> , int <i>index</i>)	Adds <i>element</i> to the vector at the location specified by <i>index</i> .
boolean isEmpty()	Returns true if the vector is empty, and returns false if it contains one or more elements.
E lastElement()	Returns the last element in the vector.
int lastIndexOf(Object <i>element</i>)	Returns the index of the last occurrence of <i>element</i> . If the object is not in the vector, -1 is returned.
int lastIndexOf(Object <i>element</i> , int <i>start</i>)	Returns the index of the last occurrence of <i>element</i> before <i>start</i> . If the object is not in that portion of the vector, -1 is returned.
void removeAllElements()	Empties the vector. After this method executes, the size of the vector is zero.
boolean removeElement(Object <i>element</i>)	Removes <i>element</i> from the vector. If more than one instance of the specified object exists in the vector, then it is the first one that is removed. Returns true if successful and false if the object is not found.
void removeElementAt(int <i>index</i>)	Removes the element at the location specified by <i>index</i> .
void setElementAt(E <i>element</i> , int <i>index</i>)	The location specified by <i>index</i> is assigned <i>element</i> .
void setSize(int <i>size</i>)	Sets the number of elements in the vector to <i>size</i> . If the new size is less than the old size, elements are lost. If the new size is larger than the old size, null elements are added.
int size()	Returns the number of elements currently in the vector.
String toString()	Returns the string equivalent of the vector.
void trimToSize()	Sets the vector's capacity equal to the number of elements that it currently holds.

Table 18-16 The Legacy Methods Defined by **Vector** (*continued*)

The following program uses a vector to store various types of numeric objects. It demonstrates several of the legacy methods defined by **Vector**. It also demonstrates the **Enumeration** interface.

```
// Demonstrate various Vector operations.
import java.util.*;

class VectorDemo {
    public static void main(String args[]) {
```

```
// initial size is 3, increment is 2
Vector<Integer> v = new Vector<Integer>(3, 2);

System.out.println("Initial size: " + v.size());
System.out.println("Initial capacity: " +
                  v.capacity());

v.addElement(1);
v.addElement(2);
v.addElement(3);
v.addElement(4);

System.out.println("Capacity after four additions: " +
                  v.capacity());

v.addElement(5);
System.out.println("Current capacity: " +
                  v.capacity());

v.addElement(6);
v.addElement(7);

System.out.println("Current capacity: " +
                  v.capacity());

v.addElement(9);
v.addElement(10);

System.out.println("Current capacity: " +
                  v.capacity());

v.addElement(11);
v.addElement(12);

System.out.println("First element: " + v.firstElement());
System.out.println("Last element: " + v.lastElement());

if(v.contains(3))
    System.out.println("Vector contains 3.");

// Enumerate the elements in the vector.
Enumeration<Integer> vEnum = v.elements();

System.out.println("\nElements in vector:");
while(vEnum.hasMoreElements())
    System.out.print(vEnum.nextElement() + " ");
System.out.println();
}

}
```

The output from this program is shown here:

```
Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5
Current capacity: 7
Current capacity: 9
First element: 1
Last element: 12
Vector contains 3.

Elements in vector:
1 2 3 4 5 6 7 9 10 11 12
```

Instead of relying on an enumeration to cycle through the objects (as the preceding program does), you can use an iterator. For example, the following iterator-based code can be substituted into the program:

```
// Use an iterator to display contents.
Iterator<Integer> vItr = v.iterator();

System.out.println("\nElements in vector:");
while(vItr.hasNext())
    System.out.print(vItr.next() + " ");
System.out.println();
```

You can also use a for-each **for** loop to cycle through a **Vector**, as the following version of the preceding code shows:

```
// Use an enhanced for loop to display contents
System.out.println("\nElements in vector:");
for(int i : v)
    System.out.print(i + " ");

System.out.println();
```

Because the **Enumeration** interface is not recommended for new code, you will usually use an iterator or a for-each **for** loop to enumerate the contents of a vector. Of course, legacy code will employ **Enumeration**. Fortunately, enumerations and iterators work in nearly the same manner.

Stack

Stack is a subclass of **Vector** that implements a standard last-in, first-out stack. **Stack** only defines the default constructor, which creates an empty stack. With the release of JDK 5, **Stack** was retrofitted for generics and is declared as shown here:

```
class Stack<E>
```

Here, **E** specifies the type of element stored in the stack.

Stack includes all the methods defined by **Vector** and adds several of its own, shown in Table 18-17.

Method	Description
boolean empty()	Returns true if the stack is empty, and returns false if the stack contains elements.
E peek()	Returns the element on the top of the stack, but does not remove it.
E pop()	Returns the element on the top of the stack, removing it in the process.
E push(E element)	Pushes <i>element</i> onto the stack. <i>element</i> is also returned.
int search(Object element)	Searches for <i>element</i> in the stack. If found, its offset from the top of the stack is returned. Otherwise, -1 is returned.

Table 18-17 The Methods Defined by **Stack**

To put an object on the top of the stack, call **push()**. To remove and return the top element, call **pop()**. You can use **peek()** to return, but not remove, the top object. An **EmptyStackException** is thrown if you call **pop()** or **peek()** when the invoking stack is empty. The **empty()** method returns **true** if nothing is on the stack. The **search()** method determines whether an object exists on the stack and returns the number of pops that are required to bring it to the top of the stack. Here is an example that creates a stack, pushes several **Integer** objects onto it, and then pops them off again:

```
// Demonstrate the Stack class.
import java.util.*;

class StackDemo {
    static void showpush(Stack<Integer> st, int a) {
        st.push(a);
        System.out.println("push(" + a + ")");
        System.out.println("stack: " + st);
    }

    static void showpop(Stack<Integer> st) {
        System.out.print("pop -> ");
        Integer a = st.pop();
        System.out.println(a);
        System.out.println("stack: " + st);
    }

    public static void main(String args[]) {
        Stack<Integer> st = new Stack<Integer>();

        System.out.println("stack: " + st);
        showpush(st, 42);
        showpush(st, 66);
        showpush(st, 99);
        showpop(st);
        showpop(st);
        showpop(st);
    }
}
```

```

        try {
            showpop(st);
        } catch (EmptyStackException e) {
            System.out.println("empty stack");
        }
    }
}

```

The following is the output produced by the program; notice how the exception handler for **EmptyStackException** is caught so that you can gracefully handle a stack underflow:

```

stack: [ ]
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: [ ]
pop -> empty stack

```

One other point: although **Stack** is not deprecated, **ArrayDeque** is a better choice.

Dictionary

Dictionary is an abstract class that represents a key/value storage repository and operates much like **Map**. Given a key and value, you can store the value in a **Dictionary** object. Once the value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs. Although not currently deprecated, **Dictionary** is classified as obsolete, because it is fully superseded by **Map**. However, **Dictionary** is still in use and thus is discussed here.

With the advent of JDK 5, **Dictionary** was made generic. It is declared as shown here:

```
class Dictionary<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values. The abstract methods defined by **Dictionary** are listed in Table 18-18.

To add a key and a value, use the **put()** method. Use **get()** to retrieve the value of a given key. The keys and values can each be returned as an **Enumeration** by the **keys()** and **elements()** methods, respectively. The **size()** method returns the number of key/value pairs stored in a dictionary, and **isEmpty()** returns **true** when the dictionary is empty. You can use the **remove()** method to delete a key/value pair.

REMEMBER The **Dictionary** class is obsolete. You should implement the **Map** interface to obtain key/value storage functionality.

Method	Purpose
Enumeration<V> elements()	Returns an enumeration of the values contained in the dictionary.
V get(Object <i>key</i>)	Returns the object that contains the value associated with <i>key</i> . If <i>key</i> is not in the dictionary, a null object is returned.
boolean isEmpty()	Returns true if the dictionary is empty, and returns false if it contains at least one key.
Enumeration<K> keys()	Returns an enumeration of the keys contained in the dictionary.
V put(K <i>key</i> , V <i>value</i>)	Inserts a key and its value into the dictionary. Returns null if <i>key</i> is not already in the dictionary; returns the previous value associated with <i>key</i> if <i>key</i> is already in the dictionary.
V remove(Object <i>key</i>)	Removes <i>key</i> and its value. Returns the value associated with <i>key</i> . If <i>key</i> is not in the dictionary, a null is returned.
int size()	Returns the number of entries in the dictionary.

Table 18-18 The Abstract Methods Defined by **Dictionary**

Hashtable

Hashtable was part of the original **java.util** and is a concrete implementation of a **Dictionary**. However, with the advent of collections, **Hashtable** was reengineered to also implement the **Map** interface. Thus, **Hashtable** is integrated into the Collections Framework. It is similar to **HashMap**, but is synchronized.

Like **HashMap**, **Hashtable** stores key/value pairs in a hash table. However, neither keys nor values can be **null**. When using a **Hashtable**, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

Hashtable was made generic by JDK 5. It is declared like this:

```
class Hashtable<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

A hash table can only store objects that override the **hashCode()** and **equals()** methods that are defined by **Object**. The **hashCode()** method must compute and return the hash code for the object. Of course, **equals()** compares two objects. Fortunately, many of Java's built-in classes already implement the **hashCode()** method. For example, the most common type of **Hashtable** uses a **String** object as the key. **String** implements both **hashCode()** and **equals()**.

The **Hashtable** constructors are shown here:

```
Hashtable()
Hashtable(int size)
Hashtable(int size, float fillRatio)
Hashtable(Map<? extends K, ? extends V> m)
```

The first version is the default constructor. The second version creates a hash table that has an initial size specified by *size*. (The default size is 11.) The third version creates a hash table that has an initial size specified by *size* and a fill ratio specified by *fillRatio*. This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash table multiplied by its fill ratio, the hash table is expanded. If you do not specify a fill ratio, then 0.75 is used. Finally, the fourth version creates a hash table that is initialized with the elements in *m*. The default load factor of 0.75 is used.

In addition to the methods defined by the **Map** interface, which **Hashtable** now implements, **Hashtable** defines the legacy methods listed in Table 18-19. Several methods throw **NullPointerException** if an attempt is made to use a **null** key or value.

Method	Description
void clear()	Resets and empties the hash table.
Object clone()	Returns a duplicate of the invoking object.
boolean contains(Object <i>value</i>)	Returns true if some value equal to <i>value</i> exists within the hash table. Returns false if the value isn't found.
boolean containsKey(Object <i>key</i>)	Returns true if some key equal to <i>key</i> exists within the hash table. Returns false if the key isn't found.
boolean containsValue(Object <i>value</i>)	Returns true if some value equal to <i>value</i> exists within the hash table. Returns false if the value isn't found.
Enumeration<V> elements()	Returns an enumeration of the values contained in the hash table.
V get(Object <i>key</i>)	Returns the object that contains the value associated with <i>key</i> . If <i>key</i> is not in the hash table, a null object is returned.
boolean isEmpty()	Returns true if the hash table is empty; returns false if it contains at least one key.
Enumeration<K> keys()	Returns an enumeration of the keys contained in the hash table.
V put(K <i>key</i> , V <i>value</i>)	Inserts a key and a value into the hash table. Returns null if <i>key</i> isn't already in the hash table; returns the previous value associated with <i>key</i> if <i>key</i> is already in the hash table.
void rehash()	Increases the size of the hash table and rehashes all of its keys.
V remove(Object <i>key</i>)	Removes <i>key</i> and its value. Returns the value associated with <i>key</i> . If <i>key</i> is not in the hash table, a null object is returned.
int size()	Returns the number of entries in the hash table.
String toString()	Returns the string equivalent of a hash table.

Table 18-19 The Legacy Methods Defined by **Hashtable**

The following example reworks the bank account program, shown earlier, so that it uses a **Hashtable** to store the names of bank depositors and their current balances:

```
// Demonstrate a Hashtable.  
import java.util.*;  
  
class HTDemo {  
    public static void main(String args[]) {  
        Hashtable<String, Double> balance =  
            new Hashtable<String, Double>();  
  
        Enumeration<String> names;  
        String str;  
        double bal;  
  
        balance.put("John Doe", 3434.34);  
        balance.put("Tom Smith", 123.22);  
        balance.put("Jane Baker", 1378.00);  
        balance.put("Tod Hall", 99.22);  
        balance.put("Ralph Smith", -19.08);  
  
        // Show all balances in hashtable.  
        names = balance.keys();  
        while(names.hasMoreElements()) {  
            str = names.nextElement();  
            System.out.println(str + ": " +  
                balance.get(str));  
        }  
  
        System.out.println();  
  
        // Deposit 1,000 into John Doe's account.  
        bal = balance.get("John Doe");  
        balance.put("John Doe", bal+1000);  
        System.out.println("John Doe's new balance: " +  
            balance.get("John Doe"));  
    }  
}
```

The output from this program is shown here:

```
Todd Hall: 99.22  
Ralph Smith: -19.08  
John Doe: 3434.34  
Jane Baker: 1378.0  
Tom Smith: 123.22  
  
John Doe's new balance: 4434.34
```

One important point: Like the map classes, **Hashtable** does not directly support iterators. Thus, the preceding program uses an enumeration to display the contents of **balance**. However, you can obtain set-views of the hash table, which permits the use of iterators. To do so, you simply use one of the collection-view methods defined by **Map**, such

as **entrySet()** or **keySet()**. For example, you can obtain a set-view of the keys and cycle through them using either an iterator or an enhanced **for** loop. Here is a reworked version of the program that shows this technique:

```
// Use iterators with a Hashtable.
import java.util.*;

class HTDemo2 {
    public static void main(String args[]) {
        Hashtable<String, Double> balance =
            new Hashtable<String, Double>();

        String str;
        double bal;

        balance.put("John Doe", 3434.34);
        balance.put("Tom Smith", 123.22);
        balance.put("Jane Baker", 1378.00);
        balance.put("Tod Hall", 99.22);
        balance.put("Ralph Smith", -19.08);

        // Show all balances in hashtable.
        // First, get a set view of the keys.
        Set<String> set = balance.keySet();

        // Get an iterator.
        Iterator<String> itr = set.iterator();
        while(itr.hasNext()) {
            str = itr.next();
            System.out.println(str + ": " +
                balance.get(str));
        }

        System.out.println();

        // Deposit 1,000 into John Doe's account.
        bal = balance.get("John Doe");
        balance.put("John Doe", bal+1000);
        System.out.println("John Doe's new balance: " +
            balance.get("John Doe"));
    }
}
```

Properties

Properties is a subclass of **Hashtable**. It is used to maintain lists of values in which the key is a **String** and the value is also a **String**. The **Properties** class is used by some other Java classes. For example, it is the type of object returned by **System.getProperties()** when obtaining environmental values. Although the **Properties** class, itself, is not generic, several of its methods are.

Properties defines the following instance variable:

Properties defaults;

This variable holds a default property list associated with a **Properties** object. **Properties** defines these constructors:

```
Properties()
Properties(Properties propDefault)
```

The first version creates a **Properties** object that has no default values. The second creates an object that uses *propDefault* for its default values. In both cases, the property list is empty.

In addition to the methods that **Properties** inherits from **Hashtable**, **Properties** defines the methods listed in Table 18-20. **Properties** also contains one deprecated method: **save()**. This was replaced by **store()** because **save()** did not handle errors correctly.

Method	Description
<code>String getProperty(String <i>key</i>)</code>	Returns the value associated with <i>key</i> . A null object is returned if <i>key</i> is neither in the list nor in the default property list.
<code>String getProperty(String <i>key</i>, String <i>defaultProperty</i>)</code>	Returns the value associated with <i>key</i> . <i>defaultProperty</i> is returned if <i>key</i> is neither in the list nor in the default property list.
<code>void list(PrintStream <i>streamOut</i>)</code>	Sends the property list to the output stream linked to <i>streamOut</i> .
<code>void list(PrintWriter <i>streamOut</i>)</code>	Sends the property list to the output stream linked to <i>streamOut</i> .
<code>void load(InputStream <i>streamIn</i>) throws IOException</code>	Inputs a property list from the input stream linked to <i>streamIn</i> .
<code>void load(Reader <i>streamIn</i>) throws IOException</code>	Inputs a property list from the input stream linked to <i>streamIn</i> .
<code>void loadFromXML(InputStream <i>streamIn</i>) throws IOException, InvalidPropertiesFormatException</code>	Inputs a property list from an XML document linked to <i>streamIn</i> .
<code>Enumeration<?> propertyNames()</code>	Returns an enumeration of the keys. This includes those keys found in the default property list, too.
<code>Object setProperty(String <i>key</i>, String <i>value</i>)</code>	Associates <i>value</i> with <i>key</i> . Returns the previous value associated with <i>key</i> , or returns null if no such association exists.
<code>void store(OutputStream <i>streamOut</i>, String <i>description</i>) throws IOException</code>	After writing the string specified by <i>description</i> , the property list is written to the output stream linked to <i>streamOut</i> .
<code>void store(Writer <i>streamOut</i>, String <i>description</i>) throws IOException</code>	After writing the string specified by <i>description</i> , the property list is written to the output stream linked to <i>streamOut</i> .
<code>void storeToXML(OutputStream <i>streamOut</i>, String <i>description</i>) throws IOException</code>	After writing the string specified by <i>description</i> , the property list is written to the XML document linked to <i>streamOut</i> .

Table 18-20 The Methods Defined by **Properties**

Method	Description
void storeToXML(OutputStream <i>streamOut</i> , String <i>description</i> , String <i>enc</i>)	The property list and the string specified by <i>description</i> is written to the XML document linked to <i>streamOut</i> using the specified character encoding.
Set<String> stringPropertyNames()	Returns a set of keys.

Table 18-20 The Methods Defined by **Properties** (*continued*)

One useful capability of the **Properties** class is that you can specify a default property that will be returned if no value is associated with a certain key. For example, a default value can be specified along with the key in the **getProperty()** method—such as **getProperty("name", "default value")**. If the "name" value is not found, then "default value" is returned. When you construct a **Properties** object, you can pass another instance of **Properties** to be used as the default properties for the new instance. In this case, if you call **getProperty("foo")** on a given **Properties** object, and "foo" does not exist, Java looks for "foo" in the default **Properties** object. This allows for arbitrary nesting of levels of default properties.

The following example demonstrates **Properties**. It creates a property list in which the keys are the names of states and the values are the names of their capitals. Notice that the attempt to find the capital for Florida includes a default value.

```
// Demonstrate a Property list.
import java.util.*;

class PropDemo {
    public static void main(String args[]) {
        Properties capitals = new Properties();

        capitals.put("Illinois", "Springfield");
        capitals.put("Missouri", "Jefferson City");
        capitals.put("Washington", "Olympia");
        capitals.put("California", "Sacramento");
        capitals.put("Indiana", "Indianapolis");

        // Get a set-view of the keys.
        Set<?> states = capitals.keySet();

        // Show all of the states and capitals.
        for(Object name : states)
            System.out.println("The capital of " +
                               name + " is " +
                               capitals.getProperty((String)name)
                               + ".");
    }
}
```

The output from this program is shown here:

The capital of Missouri is Jefferson City.
The capital of Illinois is Springfield.
The capital of Indiana is Indianapolis.
The capital of California is Sacramento.
The capital of Washington is Olympia.

The capital of Florida is Not Found.

The capital of Florida is Not Found.

Since Florida is not in the list, the default value is used.

Although it is perfectly valid to use a default value when you call `getProperty()`, as the preceding example shows, there is a better way of handling default values for most applications of property lists. For greater flexibility, specify a default property list when constructing a `Properties` object. The default list will be searched if the desired key is not found in the main list. For example, the following is a slightly reworked version of the preceding program, with a default list of states specified. Now, when Florida is sought, it will be found in the default list:

```
// Use a default property list.
import java.util.*;

class PropDemoDef {
    public static void main(String args[]) {
        Properties defList = new Properties();
        defList.put("Florida", "Tallahassee");
        defList.put("Wisconsin", "Madison");

        Properties capitals = new Properties(defList);

        capitals.put("Illinois", "Springfield");
        capitals.put("Missouri", "Jefferson City");
        capitals.put("Washington", "Olympia");
        capitals.put("California", "Sacramento");
        capitals.put("Indiana", "Indianapolis");

        // Get a set-view of the keys.
        Set<?> states = capitals.keySet();

        // Show all of the states and capitals.
        for(Object name : states)
            System.out.println("The capital of " +
                               name + " is " +
                               capitals.getProperty((String)name) +
                               ".");
    }

    System.out.println();

    // Florida will now be found in the default list.
    String str = capitals.getProperty("Florida");
    System.out.println("The capital of Florida is " +
                       + str + ".");
}
}
```

Using **store()** and **load()**

One of the most useful aspects of **Properties** is that the information contained in a **Properties** object can be easily stored to or loaded from disk with the **store()** and **load()** methods. At any time, you can write a **Properties** object to a stream or read it back. This makes property lists especially convenient for implementing simple databases. For example, the following program uses a property list to create a simple computerized telephone book that stores names and phone numbers. To find a person's number, you enter his or her name. The program uses the **store()** and **load()** methods to store and retrieve the list. When the program executes, it first tries to load the list from a file called **phonebook.dat**. If this file exists, the list is loaded. You can then add to the list. If you do, the new list is saved when you terminate the program. Notice how little code is required to implement a small, but functional, computerized phone book.

```
/* A simple telephone number database that uses
   a property list. */
import java.io.*;
import java.util.*;

class Phonebook {
    public static void main(String args[])
        throws IOException
    {
        Properties ht = new Properties();
        BufferedReader br =
            new BufferedReader(new InputStreamReader(System.in));
        String name, number;
        FileInputStream fin = null;
        boolean changed = false;

        // Try to open phonebook.dat file.
        try {
            fin = new FileInputStream("phonebook.dat");
        } catch(FileNotFoundException e) {
            // ignore missing file
        }

        /* If phonebook file already exists,
           load existing telephone numbers. */
        try {
            if(fin != null) {
                ht.load(fin);
                fin.close();
            }
        } catch(IOException e) {
            System.out.println("Error reading file.");
        }

        // Let user enter new names and numbers.
        do {
            System.out.println("Enter new name" +
                " ('quit' to stop): ");
            name = br.readLine();
            if(name.equals("quit"))
                break;
            number = br.readLine();
            ht.setProperty(name, number);
        } while(true);

        // Write the new list to the file.
        try {
            ht.store(fin, "Phonebook");
        } catch(IOException e) {
            System.out.println("Error writing file.");
        }
    }
}
```

```
name = br.readLine();
if(name.equals("quit")) continue;

System.out.println("Enter number: ");
number = br.readLine();

ht.put(name, number);
changed = true;
} while(!name.equals("quit"));

// If phone book data has changed, save it.
if(changed) {
    FileOutputStream fout = new FileOutputStream("phonebook.dat");

    ht.store(fout, "Telephone Book");
    fout.close();
}

// Look up numbers given a name.
do {
    System.out.println("Enter name to find" +
                       " ('quit' to quit): ");
    name = br.readLine();
    if(name.equals("quit")) continue;

    number = (String) ht.get(name);
    System.out.println(number);
} while(!name.equals("quit"));
}
```

Parting Thoughts on Collections

The Collections Framework gives you, the programmer, a powerful set of well-engineered solutions to some of programming's most common tasks. Consider using a collection the next time that you need to store and retrieve information. Remember, collections need not be reserved for only the “large jobs,” such as corporate databases, mailing lists, or inventory systems. They are also effective when applied to smaller jobs. For example, a **TreeMap** might make an excellent collection to hold the directory structure of a set of files. A **TreeSet** could be quite useful for storing project-management information. Frankly, the types of problems that will benefit from a collections-based solution are limited only by your imagination. One last point: In Chapter 29, the new stream API is discussed. Because streams are now integrated with collections, consider using a stream when operating on a collection.

This page has been intentionally left blank