ORACLE

Fully updated for Java SE 8 (JDK 8)

Java
The Complete Reference
Ninth Edition

Comprehensive Coverage of the Java Language

Java

Herbert Schildt

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.

The Complete Reference

Herbert Schildt

Java™

Ninth Edition

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

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.

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 xxxi

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 toString( ) . . . .
. . . . 418
Character Extraction . . . .
. 419 charAt( ) . . . .
. . . . 419 getChars( ) . . . .
. . . . 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 paint( ) . . . .
. . . . 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 java.util.concurrent . . . .
. . . . 916 java.util.concurrent.atomic 917
java.util.concurrent.locks . . . . 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 RecursiveTask<V> 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 of MenuBar, 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 Servlet Exception 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

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

The Java Language

32

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.

2 CHAPTER 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:

```
x++;
```

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

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

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 **for** 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.

3 CHAPTER 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.

Integers

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 **char**s. 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,

0x7fffffffffffffffL

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.