

Java™

The Complete Reference

Thirteenth Edition

Comprehensive Coverage of the Java Language

FULLY UPDATED AND EXPANDED



Herbert Schildt • Dr. Danny Coward



**Mc
Graw
Hill**

Java™

The Complete Reference

Thirteenth Edition

Comprehensive Coverage of the Java Language

FULLY UPDATED AND EXPANDED



Herbert Schildt • Dr. Danny Coward



**Mc
Graw
Hill**

About the Authors

Best-selling author **Herbert Schildt** has written extensively about programming for over three decades and is a leading authority on the Java language. Called “one of the world’s foremost authors of books about programming” by *International Developer* magazine, 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*, *Introducing JavaFX 8 Programming*, and *Swing: A Beginner’s Guide*. He has also written extensively about C, C++, and C#. Featured as one of the rock star programmers in Ed Burns’ book *Secrets of the Rock Star Programmers: Riding the IT Crest*, Schildt is interested in all facets of computing, but his primary focus is computer languages. Schildt holds both BA and MCS degrees from the University of Illinois.

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 helped found JavaFX technology and designed the Java WebSocket API in Java EE. 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. In addition, he is the author of two books on Java programming: *Java WebSocket Programming* and *Java EE 7: The Big Picture*. More recently, he has been applying his knowledge of Java to scale Java-based services for one of the world’s most successful software companies. Dr. Coward holds bachelor’s, master’s, and doctorate degrees in mathematics from the University of Oxford.

About the Technical Editor

Simon Ritter is the deputy CTO of Azul. Simon joined Sun Microsystems in 1996 and spent time working in both Java development and consultancy. He has been presenting Java technologies to developers since 1999, focusing on the core Java platform as well as client and embedded

applications. At Azul, he continues to help people understand Java and the JVM.

Simon is a Java Champion and two-time recipient of the JavaOne Rockstar award. In addition, he represents Azul on the JCP Executive Committee, the OpenJDK Vulnerability Group, and the JSR Expert Group since Java SE 9.

The
Complete
Reference

Java™
Thirteenth Edition

Herbert Schildt
Dr. Danny Coward



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

Copyright © 2024 by McGraw Hill. All rights reserved. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

ISBN: 978-1-26-506270-5
MHID: 1-26-506270-6

The material in this eBook also appears in the print version of this title:
ISBN: 978-1-26-505843-2, MHID: 1-26-505843-1.

eBook conversion by codeMantra
Version 1.0

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

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

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

Oracle Corporation does not make any representations or warranties as to the accuracy, adequacy, or completeness of any information contained in

this Work, and is not responsible for any errors or omissions.

TERMS OF USE

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

THE WORK IS PROVIDED “AS IS.” McGRAW HILL AND ITS LICENSORS MAKE NO GUARANTEES OR WARRANTIES AS TO THE ACCURACY, ADEQUACY OR COMPLETENESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

McGraw Hill and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither McGraw Hill nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. McGraw Hill has no responsibility for the content of any information accessed through the work. Under no circumstances shall McGraw Hill and/or its licensors be liable for any indirect, incidental, special, punitive, consequential or similar damages that result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.

Contents at a Glance

PART I The Java Language

- 1 The History and Evolution of Java
- 2 An Overview of Java
- 3 Data Types, Variables, and Arrays
- 4 Operators
- 5 Control Statements
- 6 Introducing Classes
- 7 A Closer Look at Methods and Classes
- 8 Inheritance
- 9 Packages and Interfaces
- 10 Exception Handling
- 11 Multithreaded Programming
- 12 Enumerations, Autoboxing, and Annotations
- 13 I/O, Try-with-Resources, and Other Topics
- 14 Generics
- 15 Lambda Expressions
- 16 Modules
- 17 Switch Expressions, Records, and Other Recently Added Features

PART II The Java Library

- 18 String Handling
- 19 Exploring `java.lang`
- 20 `java.util` Part 1: The Collections Framework
- 21 `java.util` Part 2: More Utility Classes
- 22 Input/Output: Exploring `java.io`
- 23 Exploring NIO
- 24 Networking
- 25 Event Handling

- 26 Introducing the AWT: Working with Windows, Graphics, and Text
- 27 Using AWT Controls, Layout Managers, and Menus
- 28 Images
- 29 The Concurrency Utilities
- 30 The Stream API
- 31 Regular Expressions and Other Packages

PART III Introducing GUI Programming with Swing

- 32 Introducing Swing
- 33 Exploring Swing
- 34 Introducing Swing Menus

PART IV Applying Java

- 35 Java Beans
- 36 Introducing Servlets

PART V Appendixes

- A Using Java's Documentation Comments
- B Introducing JShell
- C Compile and Run Simple Single-File Programs in One Step

Index

Contents

Foreword

Preface

Part I **The Java Language**

Chapter 1 **The History and Evolution of Java**

Java's Lineage

 The Birth of Modern Programming: C

 C++: The Next Step

 The Stage Is Set for Java

The Creation of Java

 The C# Connection

 Longevity

How Java Impacted the Internet

 Java Applets

 Security

 Portability

Java's Magic: The Bytecode

Moving Beyond Applets

A Faster Release Schedule

Servlets: Java on the Server Side

The Java Buzzwords

 Simple

 Object-Oriented

 Robust

 Multithreaded

 Architecture-Neutral

 Interpreted and High Performance

 Distributed

 Dynamic

The Evolution of Java

- A Culture of Innovation
- Chapter 2 An Overview of Java**
 - Object-Oriented Programming
 - Two Paradigms
 - Abstraction
 - The Three OOP Principles
 - A First Simple Program
 - Entering the Program
 - Compiling the Program
 - A Closer Look at the First Sample Program
 - A Second Short Program
 - Two Control Statements
 - The if Statement
 - The for Loop
 - Using Blocks of Code
 - Lexical Issues
 - Whitespace
 - Identifiers
 - Literals
 - Comments
 - Separators
 - The Java Keywords
 - The Java Class Libraries
- Chapter 3 Data Types, Variables, and Arrays**
 - Java Is a Strongly Typed Language
 - The Primitive Types
 - Integers
 - byte
 - short
 - int
 - long
 - Floating-Point Types
 - float
 - double

- Characters
- Booleans
- A Closer Look at Literals
 - Integer Literals
 - Floating-Point Literals
 - Boolean Literals
 - Character Literals
 - String Literals
- Variables
 - Declaring a Variable
 - Dynamic Initialization
 - The Scope and Lifetime of Variables
- Type Conversion and Casting
 - Java's Automatic Conversions
 - Casting Incompatible Types
- Automatic Type Promotion in Expressions
 - The Type Promotion Rules
- Arrays
 - One-Dimensional Arrays
 - Multidimensional Arrays
 - Alternative Array Declaration Syntax
- Introducing Type Inference with Local Variables
 - Some var Restrictions
- A Few Words About Strings

Chapter 4 Operators

- Arithmetic Operators
 - The Basic Arithmetic Operators
 - The Modulus Operator
 - Arithmetic Compound Assignment Operators
 - Increment and Decrement
- The Bitwise Operators
 - The Bitwise Logical Operators
 - The Left Shift
 - The Right Shift

- The Unsigned Right Shift
 - Bitwise Operator Compound Assignments
 - Relational Operators
 - Boolean Logical Operators
 - Short-Circuit Logical Operators
 - The Assignment Operator
 - The ? Operator
 - Operator Precedence
 - Using Parentheses
- Chapter 5 Control Statements**
- Java's Selection Statements
 - if
 - The Traditional switch
 - Iteration Statements
 - while
 - do-while
 - for
 - The For-Each Version of the for Loop
 - Local Variable Type Inference in a for Loop
 - Nested Loops
 - Jump Statements
 - Using break
 - Using continue
 - return
- Chapter 6 Introducing Classes**
- Class Fundamentals
 - The General Form of a Class
 - A Simple Class
 - Declaring Objects
 - A Closer Look at new
 - Assigning Object Reference Variables
 - Introducing Methods
 - Adding a Method to the Box Class
 - Returning a Value

Adding a Method That Takes Parameters
Constructors
Parameterized Constructors
The this Keyword
Instance Variable Hiding
Garbage Collection
A Stack Class

Chapter 7 A Closer Look at Methods and Classes

Overloading Methods
Overloading Constructors
Using Objects as Parameters
A Closer Look at Argument Passing
Returning Objects
Recursion
Introducing Access Control
Understanding static
Introducing final
Arrays Revisited
Introducing Nested and Inner Classes
Exploring the String Class
Using Command-Line Arguments
Varargs: Variable-Length Arguments

Overloading Vararg Methods

Varargs and Ambiguity

Local Variable Type Inference with Reference Types

Chapter 8 Inheritance

Inheritance Basics

Member Access and Inheritance

A More Practical Example

A Superclass Variable Can Reference a Subclass Object

Using super

Using super to Call Superclass Constructors

A Second Use for super

Creating a Multilevel Hierarchy

- When Constructors Are Executed
- Method Overriding
- Dynamic Method Dispatch
 - Why Overridden Methods?
 - Applying Method Overriding
- Using Abstract Classes
- Using final with Inheritance
 - Using final to Prevent Overriding
 - Using final to Prevent Inheritance
- Local Variable Type Inference and Inheritance
- The Object Class

Chapter 9 Packages and Interfaces

- Packages
 - Defining a Package
 - Finding Packages and CLASSPATH
 - A Short Package Example
- Packages and Member Access
 - An Access Example
- Importing Packages
- Interfaces
 - Defining an Interface
 - Implementing Interfaces
 - Nested Interfaces
 - Applying Interfaces
 - Variables in Interfaces
 - Interfaces Can Be Extended
- Default Interface Methods
 - Default Method Fundamentals
 - A More Practical Example
 - Multiple Inheritance Issues
- Use static Methods in an Interface
- Private Interface Methods
- Final Thoughts on Packages and Interfaces

Chapter 10 Exception Handling

- Exception-Handling Fundamentals
- Exception Types
- Uncaught Exceptions
- Using try and catch
 - Displaying a Description of an Exception
 - Multiple catch Clauses
 - Nested try Statements
 - throw
 - throws
 - finally
- Java's Built-in Exceptions
- Creating Your Own Exception Subclasses
- Chained Exceptions
- Three Additional Exception Features
- Using Exceptions

Chapter 11 Multithreaded Programming

- The Java Thread Model
 - Thread Priorities
 - Synchronization
 - Messaging
- The Thread Class and the Runnable Interface
- The Main Thread
- Creating a Thread
 - Implementing Runnable
 - Extending Thread
 - Choosing an Approach
- Creating Multiple Threads
- Using isAlive() and join()
- Thread Priorities
- Synchronization
 - Using Synchronized Methods
 - The synchronized Statement
- Interthread Communication
- Deadlock

- Suspending, Resuming, and Stopping Threads
- Obtaining a Thread's State
- Using a Factory Method to Create and Start a Thread
- Virtual Threads
- Using Multithreading

Chapter 12 Enumerations, Autoboxing, and Annotations

- Enumerations

- Enumeration Fundamentals
- The values() and valueOf() Methods
- Java Enumerations Are Class Types
- Enumerations Inherit Enum
- Another Enumeration Example

- Type Wrappers

- Character
- Boolean
- The Numeric Type Wrappers

- Autoboxing

- Autoboxing and Methods
- Autoboxing/Unboxing Occurs in Expressions
- Autoboxing/Unboxing Boolean and Character Values
- Autoboxing/Unboxing Helps Prevent Errors
- A Word of Warning

- Annotations

- Annotation Basics
- Specifying a Retention Policy
- Obtaining Annotations at Run Time by Use of Reflection
- The AnnotatedElement Interface
- Using Default Values
- Marker Annotations
- Single-Member Annotations
- The Built-In Annotations

- Type Annotations

- Repeating Annotations

- Some Restrictions

Chapter 13 I/O, Try-with-Resources, and Other Topics

I/O Basics

Streams

Byte Streams and Character Streams

The Predefined Streams

Reading Console Input

Reading Characters

Reading Strings

Writing Console Output

The PrintWriter Class

Reading and Writing Files

Automatically Closing a File

The transient and volatile Modifiers

Introducing instanceof

strictfp

Native Methods

Using assert

Assertion Enabling and Disabling Options

Static Import

Invoking Overloaded Constructors Through this()

A Word About Value-Based Classes

Chapter 14 Generics

What Are Generics?

A Simple Generics Example

Generics Work Only with Reference Types

Generic Types Differ Based on Their Type Arguments

How Generics Improve Type Safety

A Generic Class with Two Type Parameters

The General Form of a Generic Class

Bounded Types

Using Wildcard Arguments

Bounded Wildcards

Creating a Generic Method

Generic Constructors

- Generic Interfaces
- Raw Types and Legacy Code
- Generic Class Hierarchies
 - Using a Generic Superclass
 - A Generic Subclass
 - Run-Time Type Comparisons Within a Generic Hierarchy
- Casting
- Overriding Methods in a Generic Class
- Type Inference with Generics
- Local Variable Type Inference and Generics
- Erasure
 - Bridge Methods
- Ambiguity Errors
- Some Generic Restrictions
 - Type Parameters Can't Be Instantiated
 - Restrictions on Static Members
 - Generic Array Restrictions
 - Generic Exception Restriction

Chapter 15 Lambda Expressions

- Introducing Lambda Expressions
- Lambda Expression Fundamentals
- Functional Interfaces
- Some Lambda Expression Examples
- Block Lambda Expressions
- Generic Functional Interfaces
- Passing Lambda Expressions as Arguments
- Lambda Expressions and Exceptions
- Lambda Expressions and Variable Capture
- Method References
 - Method References to static Methods
 - Method References to Instance Methods
 - Method References with Generics
- Constructor References

Predefined Functional Interfaces

Chapter 16 Modules

Module Basics

A Simple Module Example

Compile and Run the First Module Example

A Closer Look at requires and exports

java.base and the Platform Modules

Legacy Code and the Unnamed Module

Exporting to a Specific Module

Using requires transitive

Use Services

Service and Service Provider Basics

The Service-Based Keywords

A Module-Based Service Example

Module Graphs

Three Specialized Module Features

Open Modules

The opens Statement

requires static

Introducing jlink and Module JAR Files

Linking Files in an Exploded Directory

Linking Modular JAR Files

JMOD Files

A Brief Word About Layers and Automatic Modules

Final Thoughts on Modules

Chapter 17 Switch Expressions, Records, and Other Recently Added Features

Records

Record Basics

Create Record Constructors

Another Record Constructor Example

Create Record Getter Methods

Pattern Matching with instanceof

Pattern Variables in a Logical AND Expression

Pattern Matching in Other Statements
Enhancements to switch
 Use a List of case Constants
 Introducing the switch Expression and the yield Statement
 Introducing the Arrow in a case Statement
 A Closer Look at the Arrow case
 Another switch Expression Example
 Pattern Matching in switch
Text Blocks
 Text Block Fundamentals
 Understanding Leading Whitespace
 Use Double Quotes in a Text Block
 Escape Sequences in Text Blocks
 Pattern Matching with Records
Sealed Classes and Interfaces
 Sealed Classes
 Sealed Interfaces
Future Directions

Part II The Java Library

Chapter 18 String Handling
 The String Constructors
 String Length
 Special String Operations
 String Literals
 String Concatenation
 String Concatenation with Other Data Types
 String Conversion and `toString()`
 Character Extraction
 `charAt()`
 `getChars()`
 `getBytes()`
 `toCharArray()`
 String Comparison

equals() and equalsIgnoreCase()
regionMatches()
startsWith() and endsWith()
equals() Versus ==
compareTo()

Searching Strings

Modifying a String

- substring()
- concat()
- replace()
- trim() and strip()

Data Conversion Using valueOf()

Changing the Case of Characters Within a String

Joining Strings

Additional String Methods

StringBuffer

- StringBuffer Constructors
- length() and capacity()
- ensureCapacity()
- setLength()
- charAt() and setCharAt()
- getChars()
- append()
- insert()
- reverse()
- delete() and deleteCharAt()
- replace()
- substring()

Additional StringBuffer Methods

StringBuilder

Chapter 19 Exploring java.lang

Primitive Type Wrappers

- Number
- Double and Float

Understanding isInfinite() and isNaN()
Byte, Short, Integer, and Long
Character
Additions to Character for Unicode Code Point Support
Boolean

Void

Process

Runtime

Executing Other Programs

Runtime.Version

ProcessBuilder

System

Using currentTimeMillis() to Time Program Execution

Using arraycopy()

Environment Properties

System.Logger and System.LoggerFinder

Object

Using clone() and the Cloneable Interface

Class

ClassLoader

Math

Trigonometric Functions

Exponential Functions

Rounding Functions

Miscellaneous Math Methods

StrictMath

Compiler

Thread, ThreadGroup, and Runnable

The Runnable Interface

Thread

ThreadGroup

ThreadLocal and InheritableThreadLocal

Package

Module

ModuleLayer
RuntimePermission
Throwable
SecurityManager
StackTraceElement
StackWalker and StackWalker.StackFrame
Enum
Record
ClassValue
The CharSequence Interface
The Comparable Interface
The Appendable Interface
The Iterable Interface
The Readable Interface
The AutoCloseable Interface
The Thread.UncaughtExceptionHandler Interface
The java.lang Subpackages
 java.lang.annotation
 java.lang.constant
 java.lang.instrument
 java.lang.invoke
 java.lang.management
 java.lang.module
 java.lang.ref
 java.lang.reflect

Chapter 20 java.util Part 1: The Collections Framework

Collections Overview
The Collection Interfaces
 The Collection Interface
 The SequencedCollection Interface
 The List Interface
 The Set Interface
 The SequencedSet Interface
 The SortedSet Interface

- The NavigableSet Interface
- The Queue Interface
- The Deque Interface
- The Collection Classes
 - The ArrayList Class
 - The LinkedList Class
 - The HashSet Class
 - The LinkedHashSet Class
 - The TreeSet Class
 - The PriorityQueue Class
 - The ArrayDeque Class
 - The EnumSet Class
- Accessing a Collection via an Iterator
 - Using an Iterator
 - The For-Each Alternative to Iterators
- Spliterators
- Storing User-Defined Classes in Collections
- The RandomAccess Interface
- Working with Maps
 - The Map Interfaces
 - The Map Classes
- Comparators
 - Using a Comparator
- The Collection Algorithms
- Arrays
- The Legacy Classes and Interfaces
 - The Enumeration Interface
 - Vector
 - Stack
 - Dictionary
 - Hashtable
 - Properties
 - Using store() and load()
- Parting Thoughts on Collections

Chapter 21 java.util Part 2: More Utility Classes

StringTokenizer

BitSet

Optional, OptionalDouble, OptionalInt, and OptionalLong

Date

Calendar

GregorianCalendar

TimeZone

SimpleTimeZone

Locale

Random

Timer and TimerTask

Currency

Formatter

The Formatter Constructors

The Formatter Methods

Formatting Basics

Formatting Strings and Characters

Formatting Numbers

Formatting Time and Date

The %n and %% Specifiers

Specifying a Minimum Field Width

Specifying Precision

Using the Format Flags

Justifying Output

The Space, +, 0, and (Flags

The Comma Flag

The # Flag

The Uppercase Option

Using an Argument Index

Closing a Formatter

The Java printf() Connection

Scanner

The Scanner Constructors

- Scanning Basics
- Some Scanner Examples
- Setting Delimiters
- Other Scanner Features
- The ResourceBundle, ListResourceBundle, and PropertyResourceBundle Classes
- Miscellaneous Utility Classes and Interfaces
- The java.util Subpackages
 - java.util.concurrent, java.util.concurrent.atomic, and
java.util.concurrent.locks
 - java.util.function
 - java.util.jar
 - java.util.logging
 - java.util.prefs
 - java.util.random
 - java.util.regex
 - java.util.spi
 - java.util.stream
 - java.util.zip

Chapter 22 Input/Output: Exploring java.io

- The I/O Classes and Interfaces
- File
 - Directories
 - Using FilenameFilter
 - The listFiles() Alternative
 - Creating Directories
- The AutoCloseable, Closeable, and Flushable Interfaces
- I/O Exceptions
- Two Ways to Close a Stream
- The Stream Classes
- The Byte Streams
 - InputStream
 - OutputStream
 - FileInputStream

- FileOutputStream
- ByteArrayInputStream
- ByteArrayOutputStream
- Filtered Byte Streams
- Buffered Byte Streams
- SequenceInputStream
- PrintStream
- DataOutputStream and DataInputStream
- RandomAccessFile

The Character Streams

- Reader
- Writer
- FileReader
- FileWriter
- CharArrayReader
- CharArrayWriter
- BufferedReader
- BufferedWriter
- PushbackReader
- PrintWriter

The Console Class

Serialization

- Serializable
- Externalizable
- ObjectOutput
- ObjectOutputStream
- ObjectInput
- ObjectInputStream

A Serialization Example

Stream Benefits

Chapter 23 Exploring NIO

The NIO Classes

NIO Fundamentals

Buffers

- Channels
- Charsets and Selectors
- Enhancements Added by NIO.2
 - The Path Interface
 - The Files Class
 - The Paths Class
 - The File Attribute Interfaces
 - The FileSystem, FileSystems, and FileStore Classes
- Using the NIO System
 - Use NIO for Channel-Based I/O
 - Use NIO for Stream-Based I/O
 - Use NIO for Path and File System Operations

Chapter 24 Networking

- Networking Basics
- The java.net Networking Classes and Interfaces
- InetAddress
 - Factory Methods
 - Instance Methods
- Inet4Address and Inet6Address
- TCP/IP Client Sockets
- URL
- URLConnection
- HttpURLConnection
- The URI Class
- Cookies
- TCP/IP Server Sockets
- Datagrams
 - DatagramSocket
 - DatagramPacket
 - A Datagram Example
- Introducing java.net.http
 - Three Key Elements
 - A Simple HTTP Client Example
 - Things to Explore in java.net.http

Chapter 25 Event Handling

Two Event Handling Mechanisms

The Delegation Event Model

Events

Event Sources

Event Listeners

Event Classes

The ActionEvent Class

The AdjustmentEvent Class

The ComponentEvent Class

The ContainerEvent Class

The FocusEvent Class

The InputEvent Class

The ItemEvent Class

The KeyEvent Class

The MouseEvent Class

The MouseWheelEvent Class

The TextEvent Class

The WindowEvent Class

Sources of Events

Event Listener Interfaces

The ActionListener Interface

The AdjustmentListener Interface

The ComponentListener Interface

The ContainerListener Interface

The FocusListener Interface

The ItemListener Interface

The KeyListener Interface

The MouseListener Interface

The MouseMotionListener Interface

The MouseWheelListener Interface

The TextListener Interface

The WindowFocusListener Interface

The WindowListener Interface

- Using the Delegation Event Model
- Some Key AWT GUI Concepts
- Handling Mouse Events
- Handling Keyboard Events
- Adapter Classes
- Inner Classes
- Anonymous Inner Classes

Chapter 26 Introducing the AWT: Working with Windows, Graphics, and Text

- AWT Classes
- Window Fundamentals
 - Component
 - Container
 - Panel
 - Window
 - Frame
 - Canvas
- Working with Frame Windows
 - Setting the Window's Dimensions
 - Hiding and Showing a Window
 - Setting a Window's Title
 - Closing a Frame Window
 - The paint() Method
 - Displaying a String
 - Setting the Foreground and Background Colors
 - Requesting Repainting
 - Creating a Frame-Based Application
- Introducing Graphics
 - Drawing Lines
 - Drawing Rectangles
 - Drawing Ellipses and Circles
 - Drawing Arcs
 - Drawing Polygons
 - Demonstrating the Drawing Methods

- Sizing Graphics
- Working with Color
 - Color Methods
 - Setting the Current Graphics Color
 - A Color Demonstration Program
- Setting the Paint Mode
- Working with Fonts
 - Determining the Available Fonts
 - Creating and Selecting a Font
 - Obtaining Font Information
- Managing Text Output Using FontMetrics

Chapter 27 Using AWT Controls, Layout Managers, and Menus

- AWT Control Fundamentals
 - Adding and Removing Controls
 - Responding to Controls
 - The HeadlessException
- Labels
- Using Buttons
 - Handling Buttons
- Applying Check Boxes
 - Handling Check Boxes
- CheckboxGroup
- Choice Controls
 - Handling Choice Lists
- Using Lists
 - Handling Lists
- Managing Scroll Bars
 - Handling Scroll Bars
- Using a TextField
 - Handling a TextField
- Using a TextArea
- Understanding Layout Managers
 - FlowLayout
 - BorderLayout

- Using Insets
- GridLayout
- CardLayout
- GridBagLayout
- Menu Bars and Menus
- Dialog Boxes
- A Word About Overriding `paint()`

Chapter 28 Images

- File Formats
- Image Fundamentals: Creating, Loading, and Displaying
 - Creating an Image Object
 - Loading an Image
 - Displaying an Image
- Double Buffering
- ImageProducer
 - MemoryImageSource
- ImageConsumer
- PixelGrabber
- ImageFilter
 - CropImageFilter
 - RGBImageFilter
- Additional Imaging Classes

Chapter 29 The Concurrency Utilities

- The Concurrent API Packages
 - `java.util.concurrent`
 - `java.util.concurrent.atomic`
 - `java.util.concurrent.locks`
- Using Synchronization Objects
 - Semaphore
 - CountDownLatch
 - CyclicBarrier
 - Exchanger
 - Phaser
- Using an Executor

A Simple Executor Example
Using Callable and Future
The TimeUnit Enumeration
The Concurrent Collections
Locks
Atomic Operations
Parallel Programming via the Fork/Join Framework
 The Main Fork/Join Classes
 The Divide-and-Conquer Strategy
 A Simple First Fork/Join Example
 Understanding the Impact of the Level of Parallelism
 An Example that Uses RecursiveTask<V>
 Executing a Task Asynchronously
 Cancelling a Task
 Determining a Task's Completion Status
 Restarting a Task
 Things to Explore
 Some Fork/Join Tips
The Concurrency Utilities Versus Java's Traditional Approach

Chapter 30 The Stream API

Stream Basics
 Stream Interfaces
 How to Obtain a Stream
 A Simple Stream Example

Reduction Operations

Using Parallel Streams

Mapping

Collecting

Iterators and Streams

 Use an Iterator with a Stream

 Use Spliterator

More to Explore in the Stream API

Chapter 31 Regular Expressions and Other Packages

Regular Expression Processing

Pattern
Matcher
Regular Expression Syntax
Demonstrating Pattern Matching
Two Pattern-Matching Options
Exploring Regular Expressions
Reflection
Remote Method Invocation
A Simple Client/Server Application Using RMI
Formatting Date and Time with java.text
 DateFormat Class
 SimpleDateFormat Class
The java.time Time and Date API
 Time and Date Fundamentals
 Formatting Date and Time
 Parsing Date and Time Strings
 Other Things to Explore in java.time

Part III Introducing GUI Programming with Swing

Chapter 32 Introducing Swing

The Origins of Swing
Swing Is Built on the AWT
Two Key Swing Features
 Swing Components Are Lightweight
 Swing Supports a Pluggable Look and Feel
The MVC Connection
Components and Containers
 Components
 Containers
 The Top-Level Container Panes
The Swing Packages
A Simple Swing Application
Event Handling
Painting in Swing
 Painting Fundamentals

Compute the Paintable Area A Paint Example

Chapter 33 Exploring Swing

- JLabel and ImageIcon
- JTextField
- The Swing Buttons
 - JButton
 - JToggleButton
 - Check Boxes
 - Radio Buttons
- JTabbedPane
- JScrollPane
- JList
- JComboBox
- Trees
- JTable

Chapter 34 Introducing Swing Menus

- Menu Basics
- An Overview of JMenuBar, JMenu, and JMenuItem
 - JMenuBar
 - JMenu
 - JMenuItem
- Create a Main Menu
- Add Mnemonics and Accelerators to Menu Items
- Add Images and Tooltips to Menu Items
- Use JRadioButtonMenuItem and JCheckBoxMenuItem
- Create a Popup Menu
- Create a Toolbar
- Use Actions
- Put the Entire MenuDemo Program Together
- Continuing Your Exploration of Swing

Part IV Applying Java

Chapter 35 Java Beans

- What Is a Java Bean?

Advantages of Beans

Introspection

 Design Patterns for Properties

 Design Patterns for Events

 Methods and Design Patterns

 Using the BeanInfo Interface

Bound and Constrained Properties

Persistence

Customizers

The JavaBeans API

 Introspector

 PropertyDescriptor

 EventSetDescriptor

 MethodDescriptor

A Bean Example

Chapter 36 Introducing Servlets

Background

The Life Cycle of a Servlet

Servlet Development Options

Using Tomcat

A Simple Servlet

 Create and Compile the Servlet Source Code

 Start Tomcat

 Start a Web Browser and Request the Servlet

The Servlet API

The jakarta.servlet Package

 The Servlet Interface

 The ServletConfig Interface

 The ServletContext Interface

 The ServletRequest Interface

 The ServletResponse Interface

 The GenericServlet Class

 The ServletInputStream Class

 The ServletOutputStream Class

The Servlet Exception Classes
Reading Servlet Parameters
The jakarta.servlet.http Package
 The HttpServletRequest Interface
 The HttpServletResponse Interface
 The HttpSession Interface
 The Cookie Class
 The HttpServlet Class
Handling HTTP Requests and Responses
 Handling HTTP GET Requests
 Handling HTTP POST Requests
Using Cookies
Session Tracking

Part V Appendixes

Appendix A Using Java's Documentation Comments

The javadoc Tags

@author
{@code}
@deprecated
{@docRoot}
@end
@exception
@hidden
{@index}
{@inheritDoc}
{@link}
{@linkplain}
{@literal}
@param
@provides
@return
@see
@serial
@serialData

`@serialField`
`@since`
`{@snippet}`
`@start`
`{@summary}`
`{@systemProperty}`
`@throws`
`@uses`
`{@value}`
`@version`

The General Form of a Documentation Comment

What javadoc Outputs

An Example that Uses Documentation Comments

Appendix B Introducing JShell

JShell Basics

List, Edit, and Rerun Code

Add a Method

Create a Class

Use an Interface

Evaluate Expressions and Use Built-in Variables

Importing Packages

Exceptions

Some More JShell Commands

Exploring JShell Further

Appendix C Compile and Run Simple Single-File Programs in One Step

Index

Foreword

Having written extensively about programming for nearly four decades, the time has come for me to retire. As a result, this edition of *Java: The Complete Reference* was prepared by Dr. Danny Coward. Danny is a Java expert, a published author, and the technical editor on several previous editions of this book. When it came time for me to retire, he generously agreed to take on the task of fully revising this book for JDK 21. As a result, all revisions, updates, and new material, such as coverage of **record** patterns, pattern matching for **switch**, and sequenced collections, were prepared and written by Danny. Revising a book of this size is no small task, and I want thank Danny for all his efforts in preparing this, the thirteenth edition of *Java: The Complete Reference*.

I also want to thank everyone at McGraw Hill. To become a successful writer requires a successful publisher. Each needs the other to create a quality book. (In other words, writing and publishing form two sides of the same coin.) Of course, a publisher is only as good as its people. Over these many years, I have been fortunate to have worked with some of the very best. Although there are far too many to name them all, I want to give special thanks to Wendy Rinaldi, Lisa McClain, Patty Mon, Jeff Pepper, and Scott Rogers. Friends, it's been one grand adventure and a wonderful time. All the best, and many thanks!

—HERBERT SCHILDT

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. It is also a powerful, general-purpose programming language suitable for a wide variety of purposes. 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 thirteenth edition, and it has been updated for Java SE 21 (JDK 21). As a result, this edition of the book contains a substantial amount of new material, updates, and changes. Of special interest are the discussions of the following key features that have been added to the Java platform since the previous edition of this book:

- Pattern matching in **switch**
- Record patterns
- Sequenced collections
- Virtual threads

Collectively, these constitute a substantial set of new features that significantly expand the range, scope, and expressiveness of the platform. The first two features are additions to the Java language itself. Pattern

matching in **switch** expands the expressiveness of the **switch** statement to look for values matching a number of patterns, while record patterns allows pattern matching with Java records in **instanceof** expressions for concise and powerful forms of data processing. The additional APIs in Sequenced Collections defines the idea of the *encounter order* on several of the familiar Java Collection classes, while also providing uniform APIs to manage their first and last elements and for processing their elements backwards. Virtual Threads allow developers to create Java threads in their traditional form but that are managed by Java, not by the underlying OS, thereby opening up the possibilities to scale applications that require large numbers of threads more gracefully. Collectively, these new features fundamentally expand the ways in which you can design and implement solutions.

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 four 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 including the new virtual threading. Next, it describes annotations, enumerations, autoboxing, generics, modules, and lambda expressions. I/O is also introduced. The final chapter in Part I covers several recently added features: records, sealed

classes and interfaces, the enhanced **switch**, including pattern matching, record patterns, pattern matching with **instanceof**, and text blocks.

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

Part III offers three chapters that introduce Swing.

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

Special Thanks

I thank my many mentors as a programmer and designer in Java. Bill Shannon, for teaching me to always strive for the best way to solve a problem; Graham Hamilton, for showing me what it means to be a thought leader' and James Gosling, for bringing out the magic in everyday work.

Thank you to Simon Ritter for his thoughtful and expert technical reviews for the updates to this book, both the detail and the bigger picture.

I give special thanks to Herb Schildt. In creating and maintaining this valuable and comprehensive book with such care and expertise over some 12 editions, he has helped so very many of his readers become skilled and knowledgeable Java programmers. I have enjoyed working with him as a technical reviewer for a number of past editions of this work and am grateful for his thoughtful guidance in my updates to this thirteenth edition.

DANNY COWARD

PART

I

The Java Language

CHAPTER 1

The History and Evolution of Java

CHAPTER 2

An Overview of Java

CHAPTER 3

Data Types, Variables, and Arrays

CHAPTER 4

Operators

CHAPTER 5

Control Statements

CHAPTER 6

Introducing Classes

CHAPTER 7

A Closer Look at Methods and Classes

CHAPTER 8

Inheritance

CHAPTER 9

Packages and Interfaces

CHAPTER 10

Exception Handling

CHAPTER 11

Multithreaded Programming

CHAPTER 12

Enumerations, Autoboxing, and Annotations

CHAPTER 13

I/O, Try-with-Resources, and Other Topics

CHAPTER 14

Generics

CHAPTER 15

Lambda Expressions

CHAPTER 16

Modules

CHAPTER 17

Switch Expressions, Records, and Other Recently Added Features

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 occur 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 may 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, at the time it was hardly a language that encouraged 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 at the time) 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 throughout 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.

Longevity

Even as Java has broadened its range and power, other general-purpose computer languages have been created. Some are inspired by the success of Java, often emulating some of its features, even some running on its virtual machine. Some address perceived weaknesses of Java in novel ways. These newer languages include Kotlin, which uses the same bytecode representation but aims to be simpler and easier to learn, and Ruby, a dynamically interpreted language with a straightforward syntax to allow applications to be developed quickly. Other languages include Go, Scala, and Rust. With each year that passes the list grows. But one constant in the near three decades since Java was released has been its popularity with developers. This has kept it at or near the top of the most widely used languages for computing. Perhaps this is testimony to how well Java has continued to evolve to meet the changing needs of developers and the work they do.

How Java Impacted 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

At the time of Java’s creation, one of its most exciting features was the applet. An *applet* is a special kind of Java program that is designed to be

transmitted over the Internet and automatically executed inside a Java-compatible web browser. If the user clicks a link that contains an applet, the applet will download and run in the browser. Applets were intended to be small programs. They were 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 allowed some functionality to be moved from the server to the client.

The creation of the applet was important because, at the time, it expanded the universe of objects that could 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.

In the early days of Java, applets were a crucial part of Java programming. They illustrated the power and benefits of Java, added an exciting dimension to web pages, and enabled programmers to explore the full extent of what was possible with Java. Although it is likely that there are still applets in use today, over time they became less important. For reasons that will be explained, beginning with JDK 9, the phase-out of applets began, with applet support being removed by JDK 11.

Security

As desirable as dynamic, networked programs are, they can also present serious problems in the areas of security and portability. Obviously, a program that downloads and executes 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, beginning with 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 programs to be safely downloaded and executed on the client computer, it was necessary to prevent them from launching such an attack.

Java achieved this protection by enabling you to confine an application to the Java execution environment and prevent it from accessing other parts of the computer. (You will see how this is accomplished shortly.) The ability to download programs with a degree of confidence that no harm will be done 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. In other words, a mechanism that allows the same application to be downloaded and executed by a wide variety of CPUs, operating systems, and browsers is required. It is not practical to have different versions of the application for different computers. The *same* application 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 allowed 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 what is called the *Java Virtual Machine (JVM)*, which is part of the Java Runtime Environment (JRE). 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 a JRE 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 manages program execution. Thus, it is possible for the JVM to create a restricted execution environment, called the *sandbox*, that contains the program, preventing unrestricted access to the machine. 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 an entire Java program is not compiled into executable code all at once. 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.

One other point: There has been experimentation with an *ahead-of-time* compiler for Java. Such a compiler can be used to compile bytecode into native code *prior* to execution by the JVM, rather than on-the-fly. Some previous versions of the JDK supplied an experimental ahead-of-time compiler; however, JDK 17 has removed it. Ahead-of-time compilation is a specialized feature, and it does not replace Java's traditional approach just described. Because of the highly specialized nature of ahead-of-time compilation, it is not discussed further in this book.

Moving Beyond Applets

At the time of this writing, it has been nearly three decades since Java's original release. Over those years, many changes have taken place. At the time of Java's creation, the Internet was a new and exciting innovation; web browsers were undergoing rapid development and refinement; the modern form of the smart phone had not yet been invented; and the near ubiquitous use of computers was still a few years off. As you would expect, Java has also changed and so, too, has the way that Java is used. Perhaps nothing illustrates the ongoing evolution of Java better than the applet.

As explained previously, in the early years of Java, applets were a crucial part of Java programming. They not only added excitement to a web page, they were also a highly visible part of Java, which added to its charisma. However, applets rely on a Java browser plug-in. Thus, for an applet to work, the browser must support it. Over the past few years, support for the Java browser plug-in has been waning. Simply put, without browser support, applets are not viable. Because of this, beginning with JDK 9, the phase-out of applets was begun, with support for applets being deprecated. In the language of Java, *deprecated* means that a feature is still available but flagged as obsolete. Thus, a deprecated feature should not be used for new code. The phase-out became complete with the release of JDK 11 because run-time support for applets was removed. Beginning with JDK 17, the entire Applet API was deprecated for removal.

As a point of interest, a few years after Java's creation an alternative to applets was added to Java. Called Java Web Start, it enabled an application to be dynamically downloaded from a web page. It was a deployment

mechanism that was especially useful for larger Java applications that were not appropriate for applets. The difference between an applet and a Web Start application is that a Web Start application runs on its own, not inside the browser. Thus, it looks much like a “normal” application. It does, however, require that a stand-alone JRE that supports Web Start is available on the host system. Beginning with JDK 11, Java Web Start support has been removed.

Given that neither applets nor Java Web Start are supported by modern versions of Java, you might wonder what mechanism should be used to deploy a Java application. At the time of this writing, part of the answer is to use the **jlink** tool added by JDK 9. It can create a complete run-time image that includes all necessary support for your program, including the JRE. Another part of the answer is the **jpackage** tool. Added by JDK 16, it can be used to create a ready-to-install application. Although a detailed discussion of deployment strategies is outside the scope of this book, it is something that you will want to pay close attention to going forward.

A Faster Release Schedule

Another major change has recently occurred in Java, but it does not involve changes to the language or the run-time environment. Rather, it relates to the way that Java releases are scheduled. In the past, major Java releases were typically separated by two or more years. However, subsequent to the release of JDK 9, the time between major Java releases has been decreased. Today, major releases occur every six months.

Each major release, called a *feature release*, includes those features ready at the time of the release. This increased *release cadence* enables new features and enhancements to be available to Java programmers in a timely fashion. Furthermore, it allows Java to respond quickly to the demands of an ever-changing programming environment. Simply put, the faster release schedule promises to be a very positive development for Java programmers.

In addition, feature releases include new features that are available to developers if they explicitly enable them to try them out but that have not yet been finalized. There are two types of these features. First, Preview Features are APIs and/or new language syntax that are fully designed and implemented but may yet be refined before being formally included in a future release based on feedback from developers trying them out. The

second type of such experimental features is Incubator Modules, which is a feature that is still evolving significantly and may or may not make it into a future release. Both types of “nonfinal” features give strong indications of what is to come next in feature releases.

With a cadence that is now every two years, a feature release will also be one that is supported (and thus remains viable) for a period of time longer than the six months until the next feature release. Such a release is called a long-term support release (LTS). An LTS release will be supported (and thus remain viable) for a period of time longer than six months. The first LTS release was JDK 11. The second LTS release was JDK 17 and the third JDK 21, for which this book has been updated. Because of the stability that an LTS release offers, it is likely that its feature set will define a baseline of functionality for a number of years. Consult Oracle for the latest information concerning long-term support and the LTS release schedule.

Currently, feature releases are scheduled for March and September of each year. As a result, JDK 10 was released in March 2018, which was six months after the release of JDK 9. The next release (JDK 11) was in September 2018. JDK 11 was an LTS release. This was followed by JDK 12 in March 2019, JDK 13 in September 2019, and so on. At the time of this writing, the latest release is JDK 21, which is an LTS release. Again, it is anticipated that every six months a new feature release will take place. Of course, you will want to consult the latest release schedule information.

At the time of this writing, there are a number of new Java features on the horizon. Because of the faster release schedule, it is very likely that several of them will be added to Java over the next few years. You will want to review the information and release notes provided by each six-month release in detail. It is truly an exciting time to be a Java programmer!

Servlets: Java on the Server Side

Client-side code is 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. One result was the *servlet*. A servlet is a small program that executes on the server.

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 was available through mechanisms such as CGI (Common Gateway Interface), the servlet offered 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. Today, server-side code in general constitutes a major use of Java.

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 managed 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, were 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 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 and language features, introduced anonymous and inner classes, 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 run time. 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 after 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 runtime 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:

- Breaking down large computing tasks into a number of smaller tasks that can be executed in parallel
- Automatically making use of multiple processors

Therefore, by using the Fork/Join Framework, you can design 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.

The next release of Java was Java SE 8, with the developer’s kit being called JDK 8. It has an internal version number of 1.8. JDK 8 was a 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 was, and will continue to 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 caused a new operator (the `->`) and a new syntax element to be added to the language.

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 was 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 new package was **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.

The next release of Java was Java SE 9. The developer's kit was called JDK 9. With the release of JDK 9, the internal version number is also 9. JDK 9 represented a major Java release, incorporating significant enhancements to both the Java language and its libraries. Like the JDK 5 and JDK 8 releases, JDK 9 affected the Java language and its API libraries in fundamental ways.

The primary new JDK 9 feature was *modules*, which enable you to specify the relationship and dependencies of the code that comprises an application. Modules also add another dimension to Java's access control features. The inclusion of modules caused a new syntax element and several keywords to be added to Java. Furthermore, a tool called **jlink** was added to the JDK, which enables a programmer to create a run-time image of an application that contains only the necessary modules. A new file type,

called JMOD, was created. Modules also have a profound affect on the API library because, beginning with JDK 9, the library packages are now organized into modules.

Although modules constitute a major Java enhancement, they are conceptually simple and straightforward. Furthermore, because pre-module legacy code is fully supported, modules can be integrated into the development process on your timeline. There is no need to immediately change any preexisting code to handle modules. In short, modules added substantial functionality without altering the essence of Java.

In addition to modules, JDK 9 included many other new features. One of particular interest is JShell, which is a tool that supports interactive program experimentation and learning. (An introduction to JShell is found in [Appendix B](#).) Another interesting upgrade is support for private interface methods. Their inclusion further enhanced JDK 8's support for default methods in interfaces. JDK 9 added a search feature to the **javadoc** tool and a new tag called **@index** to support it. As with previous releases, JDK 9 contained a number of enhancements to Java's API libraries.

As a general rule, in any Java release, it is the new features that receive the most attention. However, there was one high-profile aspect of Java that was deprecated by JDK 9: applets. Beginning with JDK 9, applets were no longer recommended for new projects. As explained earlier in this chapter, because of waning browser support for applets (and other factors), JDK 9 deprecated the entire applet API.

The next release of Java was Java SE 10 (JDK 10), which was released in March 2018. The primary new language feature added by JDK 10 was support for *local variable type inference*. With local variable type inference, it is now possible to let the type of a local variable be inferred from the type of its initializer, rather than being explicitly specified. To support this new capability, the context-sensitive keyword **var** was added to Java. Type inference can streamline code by eliminating the need to redundantly specify a variable's type when it can be inferred from its initializer. It can also simplify declarations in cases in which the type is difficult to discern or cannot be explicitly specified. Local variable type inference has become a common part of the contemporary programming environment. Its inclusion in Java helps keep Java up to date with evolving trends in language design. Along with a number of other changes, JDK 10 also

redefined the Java version string, changing the meaning of the version numbers so that they better align with the new time-based release schedule.

The next version of Java was Java SE 11 (JDK 11). It was released in September 2018, which was six months after JDK 10. JDK 11 was an LTS release. The primary new language feature in JDK 11 was support for the use of **var** in a lambda expression. Along with a number of tweaks and updates to the API in general, JDK 11 added a new networking API, which will be of interest to a wide range of developers. Called the *HTTP Client API*, it is packaged in **java.net.http**, and it provides enhanced, updated, and improved networking support for HTTP clients. Also, another execution mode was added to the Java launcher that enables it to directly execute simple single-file programs. JDK 11 also removed some features. Perhaps of the greatest interest because of its historical significance is the removal of support for applets. Recall that applets were first deprecated by JDK 9. With the release of JDK 11, applet support has been removed. Support for another deployment-related technology called Java Web Start was also removed from JDK 11. As the execution environment has continued to evolve, both applets and Java Web Start were rapidly losing relevance. Another key change in JDK 11 is that JavaFX was no longer included in the JDK. Instead, this GUI framework has become a separate open-source project. Because these features are no longer part of the JDK, they are not discussed in this book.

Between the JDK 11 LTS and the next LTS release (JDK 17) were five feature releases: JDK 12 through JDK 16. JDK 14 added support for the **switch** expression, which is a **switch** that produces a value and which was included in JDK 12 in preview form. Other enhancements to **switch** were also included. Text blocks, which are essentially string literals that can span more than one line, were added by JDK 15, after being previewed in JDK 13. JDK 16 enhanced **instanceof** with pattern matching and added a new type of class called a *record* along with the new context-sensitive keyword **record**. A record provides a convenient means of aggregating data. JDK 16 also supplied a new application packaging tool called **jpackage**.

JDK 17 was the second LTS release. Its major new feature was the ability to seal classes and interfaces. Sealing gives you control over the inheritance of a class and the inheritance and implementation of an interface. To this end, it adds the context-sensitive keywords **sealed**,

permits, and **non-sealed**, which is the first hyphenated Java keyword. JDK 17 marks the applet API as deprecated for removal in a future release.

At the time of this writing, Java SE 21 (JDK 21) is the latest version of Java and the third LTS Java release. Thus, it is of particular importance. Its major new features are both to the language and to the APIs. The language additions of patterns in **switch** statements and record patterns enhance the power of Java to express complex data processing procedures concisely. The API additions of Sequenced Collections bring a uniform way to deal with collections with a well-defined order in which the elements are processed, while virtual threads offer a new way for the Java platform to manage Java threads efficiently, with an API model that is entirely consistent with the existing operating system managed thread model. JDK 21 also includes several preview features, such as string templates and structured concurrency. These features are still undergoing development and may evolve further in future releases, but are available for experimentation by developers in this release.

One other point about the evolution of Java: Beginning in 2006, the process of open-sourcing Java began, which resulted in the OpenJDK project: a home for the reference implementation of Java and now also home for the ongoing evolution of Java through its formal process for proposing, refining, and integrating new features in Java, called Java Enhancement Proposals (JEP). Today, other open-source implementations of the JDK are also available. Open-sourcing further contributes to the dynamic nature of Java development. In the final analysis, Java's legacy of innovation is secure. Java remains the vibrant, nimble language that the programming world has come to expect.

The material in this book has been updated through JDK 21. Many new Java features, updates, and additions are described throughout. As the preceding discussion has highlighted, however, the history of Java programming is marked by dynamic change. You will want to review the new features in each subsequent Java release. Simply put: The evolution of Java continues!

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. Portable code 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. JDK 21 is the latest release in Java's ongoing, dynamic history.

CHAPTER

2

An Overview of Java

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

Object-Oriented Programming

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

Two Paradigms

All computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data. That is, some programs are written around “what is happening” and others are written around “who is being affected.” These are the two paradigms that govern how a program is constructed. The first

way is called the *process-oriented model*. This approach characterizes a program as a series of linear steps (that is, code). The process-oriented model can be thought of as *code acting on data*. Procedural languages such as C employ this model to considerable success. However, as mentioned in [Chapter 1](#), problems with this approach appear as programs grow larger and more complex.

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

Abstraction

An essential element of object-oriented programming is *abstraction*. Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. This abstraction allows people to use a car to drive to the grocery store without being overwhelmed by the complexity of the individual parts. 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 system you interact with through a car's dashboard controls may consist of a media system, navigation interface, and vehicle monitoring display. 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 it 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's 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](#)).

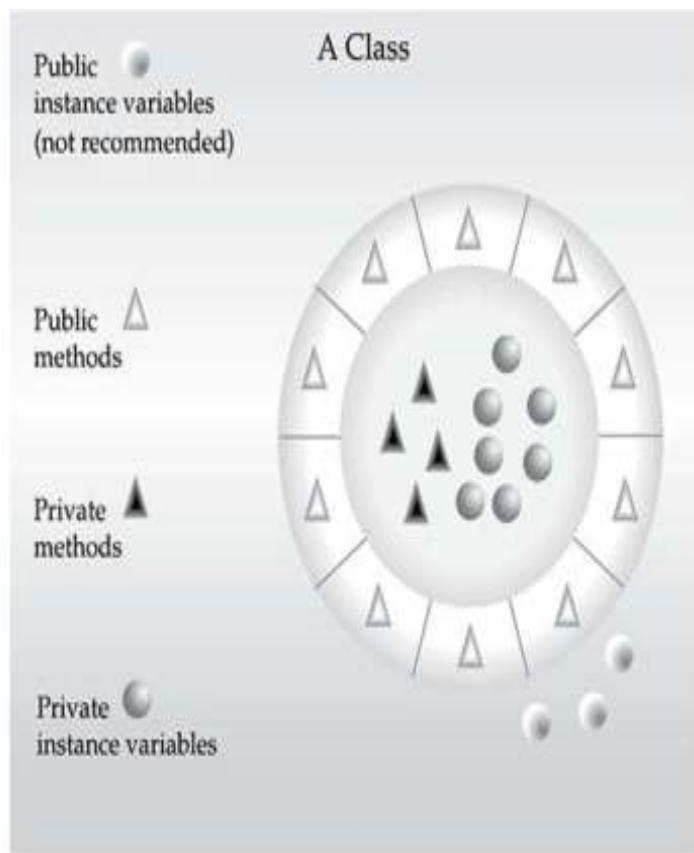


Figure 2-1 Encapsulation: public methods can be used to protect private data.

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.

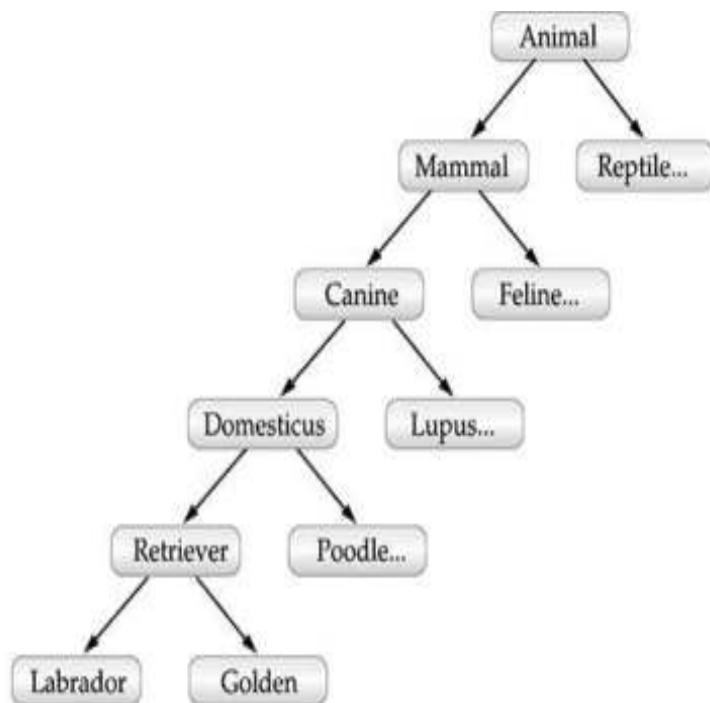
Most people naturally view the world as made up of objects that are related to each other in a hierarchical way, such as animals, mammals, and

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.

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.



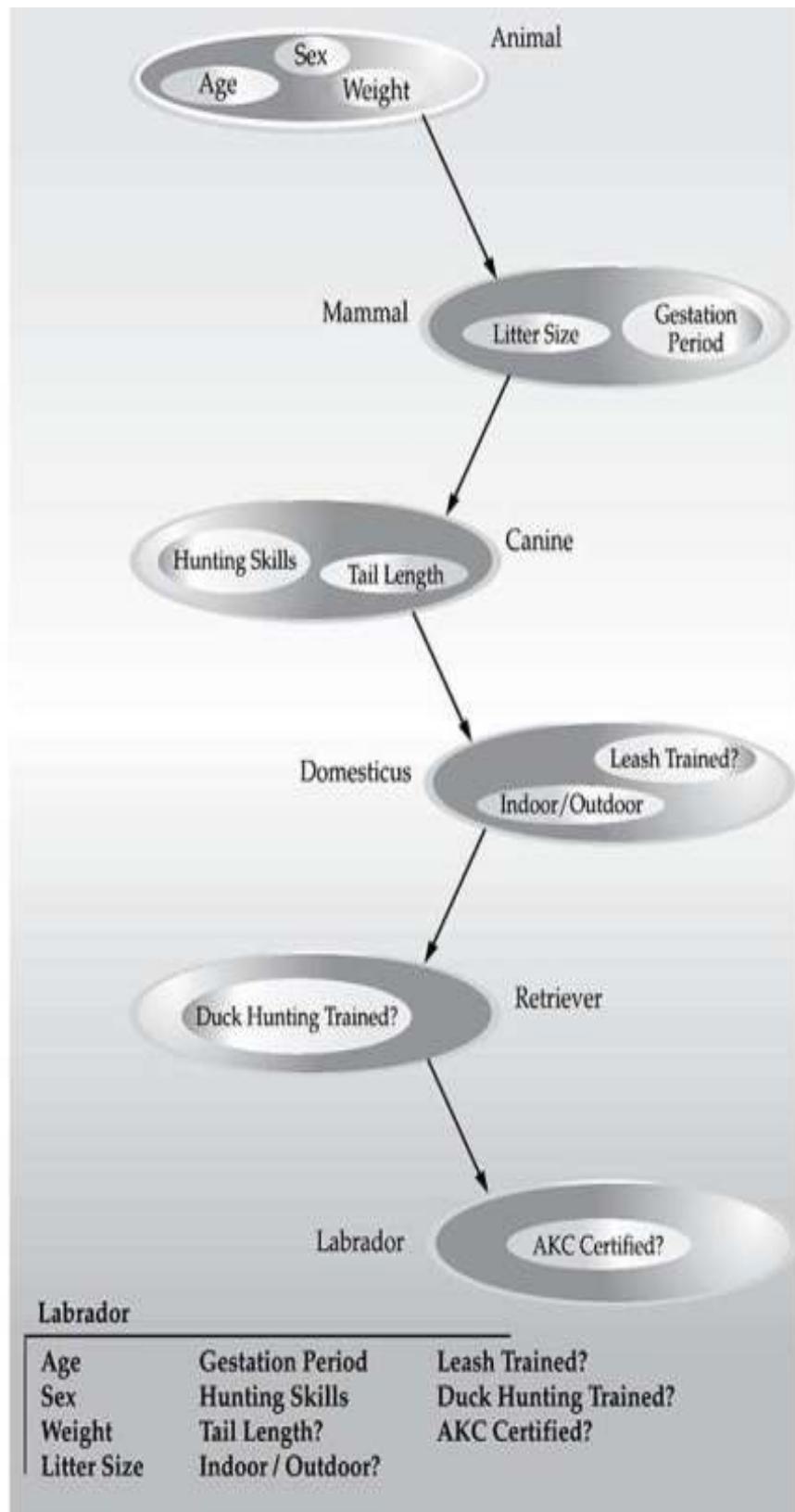


Figure 2-2 Labrador inherits the encapsulation of all its superclasses.

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 scalable 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 a 4-, 6-, or 8-cylinder engine, or an EV. 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 sample 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 Development Kit (JDK), which is available from Oracle. (Open source versions are also available.) 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 some 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. Furthermore, as you will see later in this book, in some cases, it is required.

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.

NOTE Beginning with JDK 11, Java provides a way to run some types of simple programs directly from a source file, without explicitly invoking **javac**. This technique, which can be useful in some situations, is described in [Appendix C](#). For the purposes of this book, it is assumed that you are using the normal compilation process just described.

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. As a general rule, a Java program begins 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 objects of the same type.) 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, for some types of programs, you won't need **main()** at all. However, for most of the programs shown in this book, **main()** is required.

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 that 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 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. Many statements in Java end with a semicolon. As you will see, the semicolon is an important part of the Java syntax.

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 many 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 sample 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. It determines the flow of execution based on whether some condition is true or false. Its simplest form is shown here:

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

Here, *condition* is a Boolean expression. (A Boolean expression is one that evaluates to either true or false.) 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 that 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

Loop statements are an important part of nearly any programming language because they provide a way to repeatedly execute some task. 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, *statement* executes and 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 that 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 includes a space, tab, newline, or form feed.

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 9, the underscore cannot be used by itself as an identifier.

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 [Appendix A](#).

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 often used to terminate statements. The separators are shown in the following table:

Symbol	Name	Purpose
<code>()</code>	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.
<code>{}</code>	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
<code>[]</code>	Brackets	Used to declare array types. Also used when dereferencing array values.
<code>;</code>	Semicolon	Terminates statements.
<code>,</code>	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a <code>for</code> statement.
<code>.</code>	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.
<code>::</code>	Colons	Used to create a method or constructor reference.
<code>...</code>	Ellipsis	Indicates a variable-arity parameter.
<code>@</code>	At-sign	Begins an annotation.

The Java Keywords

There are 68 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 of the Java language. The Java language limits the uses of its keywords and divides them into two categories: 51 *reserved words* and 17 *contextual keywords*. Reserved words cannot be used as identifiers, for example, variable, class, or method names. Contextual keywords have special meanings for the specific language feature that defines them but can be used elsewhere. Keywords support features added to Java over the past few years. Ten relate to modules: **exports**, **module**, **open**, **opens**, **provides**, **requires**, **to**, **transitive**, **uses**, and **with**. Records are declared by **record**; sealed classes and interfaces use **sealed**, **non-sealed**, and **permits**; **yield** and **when** are used by the enhanced **switch**; and **var** supports local variable type inference. Because keywords are context-sensitive, existing programs were unaffected by their addition. Also, beginning with JDK 9, an underscore by itself is considered a keyword in order to prevent its use as the name of something in your program. Since JDK 17, **strictfp** has been rendered obsolete because it has no effect.

abstract	assert	boolean	break	byte	case
catch	char	class	const	continue	default
do	double	else	enum	exports	extends
final	finally	float	for	goto	if
implements	import	instanceof	int	interface	long
module	native	new	non-sealed	open	opens
package	permits	private	protected	provides	public
record	requires	return	sealed	short	static
strictfp	super	switch	synchronized	this	throw
throws	to	transient	transitive	try	uses
var	void	volatile	when	while	with
yield	-				

Table 2-1 Java Keywords

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 three other names that have been part of Java from the start: **true**, **false**, and **null**. These are values defined by Java. You may not use these words for the names of variables, classes, and so on.

The Java Class Libraries

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

CHAPTER

3

Data Types, Variables, and Arrays

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

Java Is a Strongly Typed Language

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

The Primitive Types

Java defines eight *primitive* types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. The primitive types are also commonly

referred to as *simple* types, and both terms will be used in this book. These can be put in four groups:

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

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

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

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

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

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 sample **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**. A key point to understand is that 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 65,535. There are no negative **chars**. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255. Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters. Of course, the use of Unicode is somewhat inefficient for languages such as English, German, Spanish, and 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 <https://home.unicode.org>.

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

```
// Demonstrate char data type.
class CharDemo {
    public static void main(String[] args) {
        char ch1, ch2;

        ch1 = 88; // code for X
        ch2 = 'Y';

        System.out.print("ch1 and ch2: ");
        System.out.println(ch1 + " " + ch2);
    }
}
```

This program displays the following output:

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

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

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

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

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

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

The output generated by this program is shown here:

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

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

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

Booleans

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

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

```
// Demonstrate boolean values.  
class BoolTest {  
    public static void main(String[] args) {  
        boolean b;  
  
        b = false;  
        System.out.println("b is " + b);  
        b = true;  
        System.out.println("b is " + b);  
  
        // a boolean value can control the if statement  
        if(b) System.out.println("This is executed.");  
  
        b = false;  
        if(b) System.out.println("This is not executed.");
```

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

The output generated by this program is shown here:

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

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

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

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

A Closer Look at Literals

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

Integer Literals

Integers are probably the most commonly used type in the typical program. Any whole number value is an integer literal. Examples are 1, 2, 3, and 42. These are all decimal values, meaning they are describing a base 10 number. Two other bases that can be used in integer literals are *octal* (base eight) and *hexadecimal* (base 16). Octal values are denoted in Java by a leading zero. Normal decimal numbers cannot have a leading zero. Thus, the seemingly valid value 09 will produce an error from the compiler, since

9 is outside of octal's 0 to 7 range. A more common base for numbers used by programmers is hexadecimal, which matches cleanly with modulo 8 word sizes, such as 8, 16, 32, and 64 bits. You signify a hexadecimal constant with a leading zero-x (**0x** or **0X**). The range of a hexadecimal digit is 0 to 15, so *A* through *F* (or *a* through *f*) are substituted for 10 through 15.

Integer literals create an **int** value, which in Java is a 32-bit integer value. Since Java is strongly typed, you might be wondering how it is possible to assign an integer literal to one of Java's other integer types, such as **byte** or **long**, without causing a type mismatch error. Fortunately, such situations are easily handled. When a literal value is assigned to a **byte** or **short** variable, no error is generated if the literal value is within the range of the target type. An integer literal can always be assigned to a **long** variable. However, to specify a **long** literal, you will need to explicitly tell the compiler that the literal value is of type **long**. You do this by appending an upper- or lowercase *L* to the literal. For example, `0x7fffffffffffffL` or `9223372036854775807L`, is the largest **long**. An integer can also be assigned to a **char** as long as it is within range.

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.

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.

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.

Escape Sequence	Description
\ddd	Octal character (ddd)
\xxxxx	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
\s	Space (added by JDK 15)
\endofline	Continue line (applies only to text blocks; added by JDK 15)

Table 3-1 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

```
"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 string literals is that they must begin and end on the same line, even if the line wraps. For string literals there is no line-continuation escape sequence as there is in some other languages. (It is useful to point out that beginning with JDK 15, Java added a feature called a *text block*, which gives you more control and flexibility when multiple lines of text are needed. See [Chapter 17](#).)

NOTE As you may know, in some other languages 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.

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

The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope. A method's scope ends with its closing curly brace. This block of code is called the *method body*.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The output generated by this program is shown here:

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

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

lost.

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

```
// This program will not compile
class ScopeErr {
    public static void main(String[] args) {
        int bar = 1;
        {
            int bar = 2; // creates a new scope
            int bar = 2; // Compile-time error - bar already defined!
        }
    }
}
```

Type Conversion and Casting

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

Java's Automatic Conversions

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

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

When these two conditions are met, a *widening conversion* takes place. For example, the **int** type is always large enough to hold all valid **byte**

values, so no explicit cast statement is required.

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

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

Casting Incompatible Types

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

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

(target-type) value

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

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

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

is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

The following program demonstrates some type conversions that require casts:

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

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

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

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

This program generates the following output:

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

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

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

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

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

Automatic Type Promotion in Expressions

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

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

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

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

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

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

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

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

which yields the correct value of 100.

The Type Promotion Rules

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

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

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

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

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

In the first subexpression, **f * b**, **b** is promoted to a **float** and the result of the subexpression is **float**. Next, in the subexpression **i/c**, **c** is promoted to

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

Arrays

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

One-Dimensional Arrays

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

type[] var-name;

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

```
int[] month_days;
```

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

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

array-var = new type [size];

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

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

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

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

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

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

The next line displays the value stored at index 3:

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

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

```
// Demonstrate a one-dimensional array.  
class Array {  
    public static void main(String[] args) {  
        int[] month_days;  
        month_days = new int[12];  
        month_days[0] = 31;  
        month_days[1] = 28;  
        month_days[2] = 31;  
        month_days[3] = 30;  
        month_days[4] = 31;  
        month_days[5] = 30;  
        month_days[6] = 31;  
        month_days[7] = 31;  
        month_days[8] = 30;  
        month_days[9] = 31;  
        month_days[10] = 30;  
        month_days[11] = 31;  
        System.out.println("April has " + month_days[3] + " days.");  
    }  
}
```

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

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

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

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

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

```

// An improved version of the previous program.
class AutoArray {
    public static void main(String[] args) {

        int[] month_days = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
                            30, 31 };
        System.out.println("April has " + month_days[3] + " days.");
    }
}

```

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

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

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

```

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

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

```

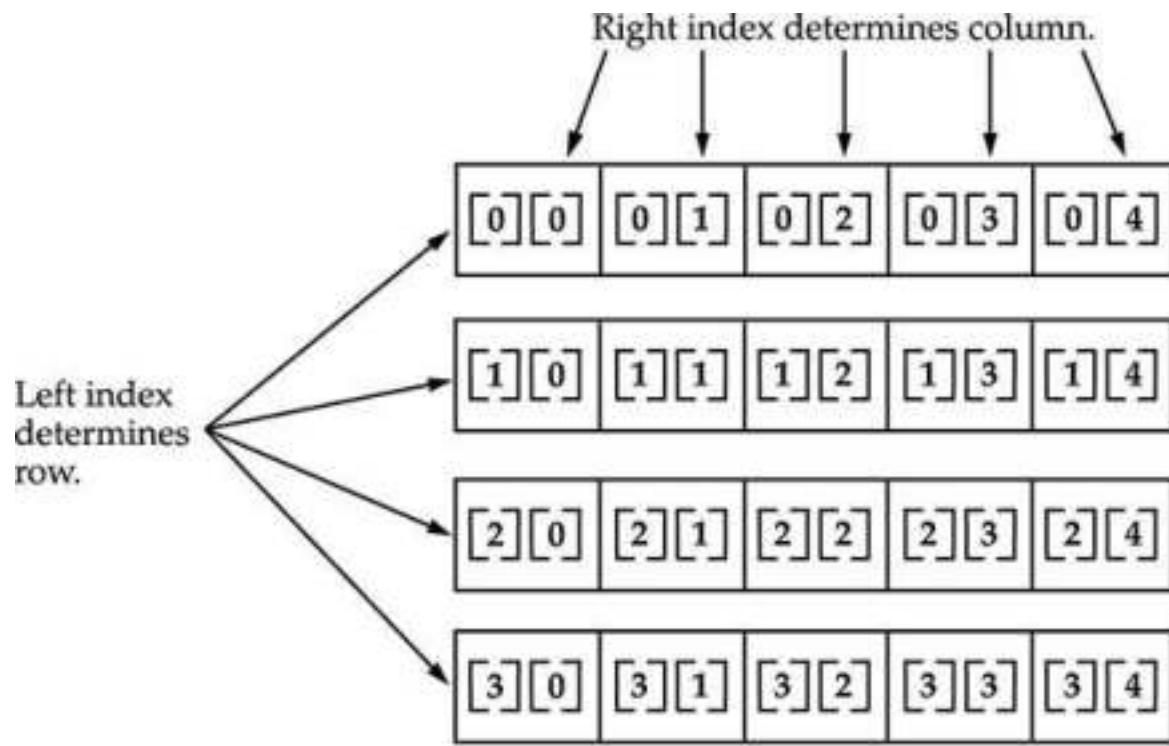
Multidimensional Arrays

In Java, *multidimensional arrays* are implemented as arrays of arrays. To declare a multidimensional array variable, specify each additional index

using another set of square brackets. For example, the following declares a two-dimensional array variable called **twoD**:

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

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



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

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

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

```
// Demonstrate a two-dimensional array.  
class TwoDArray {  
    public static void main(String[] args) {  
        int[][] twoD = new int[4][5];  
        int i, j, k = 0;  
  
        for(i=0; i<4; i++)  
            for(j=0; j<5; j++) {  
                twoD[i][j] = k;  
                k++;  
            }  
  
        for(i=0; i<4; i++) {  
            for(j=0; j<5; j++)  
                System.out.print(twoD[i][j] + " ");  
            System.out.println();  
        }  
    }  
}
```

This program generates the following output:

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

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension separately.

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

While there is no advantage to individually allocating the second dimension arrays in this situation, there may be in others. For example,

when you allocate dimensions individually, you do not need to allocate the same number of elements for each dimension. As stated earlier, since multidimensional arrays are actually arrays of arrays, the length of each array is under your control. For example, the following program creates a two-dimensional array in which the sizes of the second dimension are unequal:

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

        int i, j, k = 0;

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

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

This program generates the following output:

```
0
1 2
3 4 5
6 7 8 9
```

The array created by this program looks like this:

[0][0]			
[1][0]	[1][1]		
[2][0]	[2][1]	[2][2]	
[3][0]	[3][1]	[3][2]	[3][3]

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

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

```
// Initialize a two-dimensional array.
class Matrix {
    public static void main(String[] args) {
```

```
double[][] m = {
    { 0*0, 1*0, 2*0, 3*0 },
    { 0*1, 1*1, 2*1, 3*1 },
    { 0*2, 1*2, 2*2, 3*2 },
    { 0*3, 1*3, 2*3, 3*3 }
};
int i, j;

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

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

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

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

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

```

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

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

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

```

This program generates the following output:

```

0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

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

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

```

Alternative Array Declaration Syntax

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

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

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

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

The following declarations are also equivalent:

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

This alternative declaration form offers convenience when converting code from C/C++ to Java. It also lets you declare both array and non-array variables in a single declaration statement. Today, the alternative form of array declaration is less commonly used, but it is still important that you are familiar with it because both forms of array declaration are legal in Java.

Introducing Type Inference with Local Variables

Not long ago, a new feature called *local variable type inference* was added to the Java language. To begin, let's review two important aspects of variables. First, all variables in Java must be declared prior to their use. Second, a variable can be initialized with a value when it is declared. Furthermore, when a variable is initialized, the type of the initializer must be the same as (or convertible to) the declared type of the variable. Thus, in principle, it would not be necessary to specify an explicit type for an initialized variable because it could be inferred by the type of its initializer. Of course, in the past, such inference was not supported, and all variables required an explicitly declared type, whether they were initialized or not. Today, that situation has changed.

Beginning with JDK 10, it is now possible to let the compiler infer the type of a local variable based on the type of its initializer, thus avoiding the need to explicitly specify the type. Local variable type inference offers a number of advantages. For example, it can streamline code by eliminating

the need to redundantly specify a variable's type when it can be inferred from its initializer. It can simplify declarations in cases in which the type name is quite lengthy, such as can be the case with some class names. It can also be helpful when a type is difficult to discern or cannot be denoted. (An example of a type that cannot be denoted is the type of an anonymous class, discussed in [Chapter 25](#).) Furthermore, local variable type inference has become a common part of the contemporary programming environment. Its inclusion in Java helps keep Java up to date with evolving trends in language design. To support local variable type inference, the context-sensitive keyword **var** was added.

To use local variable type inference, the variable must be declared with **var** as the type name and it must include an initializer. For example, in the past you would declare a local **double** variable called **avg** that is initialized with the value 10.0 as shown here:

```
double avg = 10.0;
```

Using type inference, this declaration can now also be written like this:

```
var avg = 10.0;
```

In both cases, **avg** will be of type **double**. In the first case, its type is explicitly specified. In the second, its type is inferred as **double** because the initializer 10.0 is of type **double**.

As mentioned, **var** is context-sensitive. When it is used as the type name in the context of a local variable declaration, it tells the compiler to use type inference to determine the type of the variable being declared based on the type of the initializer. Thus, in a local variable declaration, **var** is a placeholder for the actual, inferred type. However, when used in most other places, **var** is simply a user-defined identifier with no special meaning. For example, the following declaration is still valid:

```
int var = 1; // In this case, var is simply a user-defined identifier.
```

In this case, the type is explicitly specified as **int** and **var** is the name of the variable being declared. Even though it is context-sensitive, there are a few

places in which the use of **var** is illegal. It cannot be used as the name of a class, for example.

The following program puts the preceding discussion into action:

```
// A simple demonstration of local variable type inference.
class VarDemo {
    public static void main(String[] args) {

        // Use type inference to determine the type of the
        // variable named avg. In this case, double is inferred.
        var avg = 10.0;
        System.out.println("Value of avg: " + avg);

        // In the following context, var is not a predefined identifier.
        // It is simply a user-defined variable name.
        int var = 1;
        System.out.println("Value of var: " + var);

        // Interestingly, in the following sequence, var is used
        // as both the type of the declaration and as a variable name
        // in the initializer.
        var k = -var;
        System.out.println("Value of k: " + k);
    }
}
```

Here is the output:

```
Value of avg: 10.0
Value of var: 1
Value of k: -1
```

The preceding example uses **var** to declare only simple variables, but you can also use **var** to declare an array. For example:

```
var myArray = new int[10]; // This is valid.
```

Notice that neither **var** nor **myArray** has brackets. Instead, the type of **myArray** is inferred to be **int[]**. Furthermore, you *cannot* use brackets on

the left side of a **var** declaration. Thus, both of these declarations are invalid:

```
var[] myArray = new int[10]; // Wrong  
var myArray[] = new int[10]; // Wrong
```

In the first line, an attempt is made to bracket **var**. In the second, an attempt is made to bracket **myArray**. In both cases, the use of the brackets is wrong because the type is inferred from the type of the initializer.

It is important to emphasize that **var** can be used to declare a variable only when that variable is initialized. For example, the following statement is incorrect:

```
var counter; // Wrong! Initializer required.
```

Also, remember that **var** can be used only to declare local variables. It cannot be used when declaring instance variables, parameters, or return types, for example.

Although the preceding discussion and examples have introduced the basics of local variable type inference, they haven't shown its full power. As you will see in [Chapter 7](#), local variable type inference is especially effective in shortening declarations that involve long class names. It can also be used with generic types (see [Chapter 14](#)), in a **try-with-resources** statement (see [Chapter 13](#)), and with a **for** loop (see [Chapter 5](#)).

Some var Restrictions

In addition to those mentioned in the preceding discussion, several other restrictions apply to the use of **var**. Only one variable can be declared at a time; a variable cannot use **null** as an initializer; and the variable being declared cannot be used by the initializer expression. Although you can declare an array type using **var**, you cannot use **var** with an array initializer. For example, this is valid:

```
var myArray = new int[10]; // This is valid.
```

However, this is not:

```
var myArray = { 1, 2, 3 }; // Wrong
```

As mentioned earlier, **var** cannot be used as the name of a class. It also cannot be used as the name of other reference types, including an interface, enumeration, or annotation, or as the name of a generic type parameter, all of which are described later in this book. Here are two other restrictions that relate to Java features described in subsequent chapters but mentioned here in the interest of completeness. Local variable type inference cannot be used to declare the exception type caught by a **catch** statement. Also, neither lambda expressions nor method references can be used as initializers.

NOTE At the time of this writing, a number of readers of this book will be using Java environments that don't support local variable type inference. So that as many of the code examples as possible will compile and run for all readers, local variable type inference will not be used by most of the programs in the remainder of this edition of the book. Using the full declaration syntax also makes it very clear at a glance what type of variable is being created, which is important for the sample code. Of course, going forward, you should consider the use of local variable type inference where appropriate in your own code.

A Few Words About Strings

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

The **String** type is used to declare string variables. You can also declare arrays of strings. A quoted string constant can be assigned to a **String** variable. A variable of type **String** can be assigned to another variable of type **String**. You can use an object of type **String** as an argument to **println()**. For example, consider the following fragment:

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

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

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

CHAPTER

4

Operators

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

Arithmetic Operators

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

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

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

The Basic Arithmetic Operators

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

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

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

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

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

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

Floating Point Arithmetic
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5
```

The Modulus Operator

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

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

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

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

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

Arithmetic Compound Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation with an assignment. As you probably know, statements like the

following are quite common in programming:

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

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

```
a += 4;
```

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

Here is another example,

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

which can be expressed as

```
a %= 2;
```

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

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

var = *var op expression*;

can be rewritten as

var op= expression;

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

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

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

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

The output of this program is shown here:

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

Increment and Decrement

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

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

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

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

```
x++;
```

Similarly, this statement:

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

is equivalent to

```
x--;
```

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

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

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

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

However, when written like this,

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

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

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

The following program demonstrates the increment operator:

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

The output of this program follows:

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

The Bitwise Operators

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

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

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

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

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

represented by inverting all of the bits in 42, or 00101010, which yields 11010101, then adding 1, which results in 11010110, or -42. To decode a negative number, first invert all of the bits, then add 1. For example, -42, or 11010110 inverted, yields 00101001, or 41, so when you add 1 you get 42.

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

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

The Bitwise Logical Operators

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

A	B	A B	A&B	A^B	$\sim A$
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

The Bitwise NOT

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

00101010

becomes

11010101

after the NOT operator is applied.

The Bitwise AND

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

$$\begin{array}{r}
 00101010 \quad 42 \\
 \&00001111 \quad 15 \\
 \hline
 00001010 \quad 10
 \end{array}$$

The Bitwise OR

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

00101010	42
00001111	15
<hr/>	
00101111	47

The Bitwise XOR

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

00101010	42
<code>^</code> 00001111	15
<hr/>	
00100101	37

Using the Bitwise Logical Operators

The following program demonstrates the bitwise logical operators:

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

```

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

```

In this example, **a** and **b** have bit patterns that present all four possibilities for two binary digits: 0-0, 0-1, 1-0, and 1-1. You can see how the **|** and **&** operate on each bit by the results in **c** and **d**. The values assigned to **e** and **f** are the same and illustrate how the **^** works. The string array named **binary** holds the human-readable, binary representation of the numbers 0 through 15. In this example, the array is indexed to show the binary representation of each result. The array is constructed such that the correct string representation of a binary value **n** is stored in **binary[n]**. The value of **~a** is ANDed with **0x0f** (0000 1111 in binary) in order to reduce its value to less than 16, so it can be printed by use of the **binary** array. Here is the output from this program:

```

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

```

The Left Shift

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

value << num

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

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

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

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

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

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

The output generated by this program is shown here:

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

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

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

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

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

The program generates the following output:

```
536870908
1073741816
2147483632
-32
```

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

The Right Shift

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

value >> *num*

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

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

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

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

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

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

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

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

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

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

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

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

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

        byte b = (byte) 0xf1;

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

Here is the output of this program:

```
b = 0xf1
```

The Unsigned Right Shift

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

you will use Java's unsigned, shift-right operator, `>>>`, which always shifts zeros into the high-order bit.

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

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

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

11111111 11111111 11111111 11111111	–1 in binary as an int
<code>>>>24</code>	
00000000 00000000 00000000 11111111	255 in binary as an int

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

```

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

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

```

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

```

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

```

Bitwise Operator Compound Assignments

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

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

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

```
a = a | b;  
a |= b;
```

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

```
class OpBitEquals {  
    public static void main(String[] args) {  
        int a = 1;  
        int b = 2;  
        int c = 3;  
  
        a |= 4;  
        b >>= 1;  
        c <<= 1;  
        a ^= c;  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
        System.out.println("c = " + c);  
    }  
}
```

The output of this program is shown here:

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

Relational Operators

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

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

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

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

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

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

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

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

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

In Java, these statements must be written like this:

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

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

Boolean Logical Operators

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

Operator	Result
&	Logical AND
	Logical OR
^	Logical XOR (exclusive OR)
	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
=	OR assignment
^=	XOR assignment

Operator	Result
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code>?:</code>	Ternary if-then-else

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

A	B	<code>A B</code>	<code>A&B</code>	<code>A^B</code>	<code>!A</code>
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

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

```

// Demonstrate the boolean logical operators.
class BoolLogic {
    public static void main(String[] args) {
        boolean a = true;
        boolean b = false;
        boolean c = a | b;
        boolean d = a & b;
        boolean e = a ^ b;
        boolean f = (!a & b) | (a & !b);
        boolean g = !a;
        System.out.println("      a = " + a);
        System.out.println("      b = " + b);
        System.out.println("      a|b = " + c);
        System.out.println("      a&b = " + d);
        System.out.println("      a^b = " + e);
        System.out.println("!a&b|a&!b = " + f);
        System.out.println("      !a = " + g);
    }
}

```

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

```

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

```

Short-Circuit Logical Operators

Java provides two interesting Boolean operators not found in some other computer languages. These are secondary versions of the Boolean AND and OR operators, and are commonly known as *short-circuit* logical operators. As you can see from the preceding table, the OR operator results in **true** when **A** is **true**, no matter what **B** is. Similarly, the AND operator

results in **false** when **A** is **false**, no matter what **B** is. If you use the `||` and `&&` forms rather than the `|` and `&` forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone. This is very useful when the right-hand operand depends on the value of the left one in order to function properly. For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

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

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

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

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

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

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

The Assignment Operator

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

var = *expression*;

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

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

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

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

The ? Operator

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

$$\textit{expression1} \ ? \ \textit{expression2} : \textit{expression3}$$

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

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

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

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

entire ? expression. The result produced by the ? operator is then assigned to **ratio**.

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

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

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

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

The output generated by the program is shown here:

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

Operator Precedence

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

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

Table 4-1 The Precedence of the Java Operators

Using Parentheses

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

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

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

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

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

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

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

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

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

CHAPTER

5

Control Statements

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

Java's Selection Statements

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

if

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
waitForMoreData();
bytesAvailable = n;
}
```

Nested ifs

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

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

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

The if-else-if Ladder

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

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

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

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

```
// Demonstrate if-else-if statements.
class IfElse {
    public static void main(String[] args) {
```

```

int month = 4; // April
String season;

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

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

```

Here is the output produced by the program:

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

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

The Traditional switch

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

At the outset, it is necessary to state that beginning with JDK 14, the **switch** has been significantly enhanced and expanded with several new features that go far beyond its traditional form. The traditional form of **switch** has been part of Java from the beginning and is, therefore, in widespread use. Furthermore, it is the form that will work in all Java development environments and for all readers. Because of the substantial nature of the recent **switch** enhancements, they are described in [Chapter 17](#),

in the context of other recent additions to Java. Here, the traditional form of the **switch** is examined. Here is the general form of a traditional **switch** statement:

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

For versions of Java prior to JDK 7, *expression* must resolve to type **byte**, **short**, **int**, **char**, or an enumeration. (Enumerations are described in [Chapter 12](#).) Today, *expression* can also be of type **String**, and beginning with JDK 21, in some situations, it can be an object reference (see [Chapter 17](#)). Each value specified in the **case** statements must be a unique constant expression (such as a literal value). Duplicate **case** values are not allowed. The type of each value must be compatible with the type of *expression*.

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

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

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

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

The output produced by this program is shown here:

```
i is zero.
i is one.
i is two.

i is three.
i is greater than 3.
i is greater than 3.
```

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

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

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

This program generates the following output:

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

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

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

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

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

As mentioned, you can also use a string to control a **switch** statement. For example:

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

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

        String str = "two";

        switch(str) {
            case "one":
                System.out.println("one");
                break;
            case "two":
                System.out.println("two");
                break;
            case "three":

                System.out.println("three");
                break;
            default:
                System.out.println("no match");
                break;
        }
    }
}
```

As you would expect, the output from the program is

```
two
```

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

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

Nested switch Statements

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

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

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

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

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

The last point is particularly interesting because it gives insight into how the Java compiler works. When it compiles a **switch** statement, the Java

compiler will inspect each of the **case** constants and create a “jump table” that it will use for selecting the path of execution depending on the value of the expression. Therefore, if you need to select among a large group of values, a **switch** statement will run much faster than the equivalent logic coded using a sequence of **if-elses**. The compiler can do this because it knows that the **case** constants are all the same type and simply must be compared for equality with the **switch** expression. The compiler has no such knowledge of a long list of **if** expressions.

REMEMBER Recently, the capabilities and features of **switch** have been substantially expanded beyond those offered by the traditional **switch** just described. Refer to [Chapter 17](#) for details on the enhanced **switch**.

Iteration Statements

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

while

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

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

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

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

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

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

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

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

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

```
int a = 10, b = 20;

while(a > b)
    System.out.println("This will not be displayed");
```

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

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

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

```
Midpoint is 150
```

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

do-while

As you just saw, if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately,

Java supplies a loop that does just that: the **do-while**. The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

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

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

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

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

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

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

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

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

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

        do {
            System.out.println("Help on: ");
            System.out.println(" 1. if");
            System.out.println(" 2. switch");
            System.out.println(" 3. while");
            System.out.println(" 4. do-while");
            System.out.println(" 5. for\n");
            System.out.println("Choose one:");
            choice = (char) System.in.read();
        } while( choice < '1' || choice > '5');

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

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

Here is a sample run produced by this program:

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

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

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

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

for

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

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

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

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

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

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

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

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

Declaring Loop Control Variables Inside the **for** Loop

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

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

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

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

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

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

        num = 14;
```

```

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

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

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

```

Using the Comma

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

```

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

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

```

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

multiple statements in both the initialization and iteration portions of the **for**. Each statement is separated from the next by a comma.

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

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

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

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

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

Some for Loop Variations

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

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

```
boolean done = false;

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

In this example, the **for** loop continues to run until the **boolean** variable **done** is set to **true**. It does not test the value of **i**.

Here is another interesting **for** loop variation. Either the initialization or the iteration expression or both may be absent, as in this next program:

```
// Parts of the for loop can be empty.
class ForVar {
    public static void main(String[] args) {
        int i;
        boolean done = false;

        i = 0;
        for( ; !done; ) {
            System.out.println("i is " + i);
            if(i == 10) done = true;
            i++;
        }
    }
}
```

Here, the initialization and iteration expressions have been moved out of the **for**. Thus, parts of the **for** are empty. While this is of no value in this simple example—indeed, it would be considered quite poor style—there can be times when this type of approach makes sense. For example, if the initial condition is set through a complex expression elsewhere in the program or if the loop control variable changes in a nonsequential manner determined by actions that occur within the body of the loop, it may be appropriate to leave these parts of the **for** empty.

Here is one more **for** loop variation. You can intentionally create an infinite loop (a loop that never terminates) if you leave all three parts of the **for** empty. For example:

```
for( ; ; ) {  
    // ...  
}
```

This loop will run forever because there is no condition under which it will terminate. Although there are some programs, such as operating system command processors, that require an infinite loop, most “infinite loops” are really just loops with special termination requirements. As you will soon see, there is a way to terminate a loop—even an infinite loop like the one shown—that does not make use of the normal loop conditional expression.

The For-Each Version of the **for** Loop

A second form of **for** implements a “for-each” style loop. As you may know, contemporary language theory has embraced the for-each concept, and it has become a standard feature that programmers have come to expect. A for-each style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. In Java, the for-each style of **for** is also referred to as the *enhanced for* loop.

The general form of the for-each version of the **for** is shown here:

for(type itr-var : collection) statement-block

Here, *type* specifies the type and *itr-var* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by *collection*. There are various types of collections that can be used with the **for**, but the only type used in this chapter is the array. (Other types of collections that can be used with the **for**, such as those defined by the Collections Framework, are discussed later in this book.) With each iteration of the loop, the next element in the collection is retrieved and stored in *itr-var*. The loop repeats until all elements in the collection have been obtained.

Because the iteration variable receives values from the collection, *type* must be the same as (or compatible with) the elements stored in the collection. Thus, when iterating over arrays, *type* must be compatible with the element type of the array.

To understand the motivation behind a for-each style loop, consider the type of **for** loop that it is designed to replace. The following fragment uses a traditional **for** loop to compute the sum of the values in an array:

```
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
int sum = 0;  
  
for(int i=0; i < 10; i++) sum += nums[i];
```

To compute the sum, each element in **nums** is read, in order, from start to finish. Thus, the entire array is read in strictly sequential order. This is accomplished by manually indexing the **nums** array by **i**, the loop control variable.

The for-each style **for** automates the preceding loop. Specifically, it eliminates the need to establish a loop counter, specify a starting and ending value, and manually index the array. Instead, it automatically cycles through the entire array, obtaining one element at a time, in sequence, from beginning to end. For example, here is the preceding fragment rewritten using a for-each version of the **for**:

```
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
int sum = 0;  
  
for(int x: nums) sum += x;
```

With each pass through the loop, **x** is automatically given a value equal to the next element in **nums**. Thus, on the first iteration, **x** contains 1; on the second iteration, **x** contains 2; and so on. Not only is the syntax streamlined, but it also prevents boundary errors.

Here is an entire program that demonstrates the for-each version of the **for** just described:

```
// Use a for-each style for loop.
class ForEach {
    public static void main(String[] args) {
        int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int sum = 0;

        // use for-each style for to display and sum the values
        for(int x : nums) {
            System.out.println("Value is: " + x);
            sum += x;
        }

        System.out.println("Summation: " + sum);
    }
}
```

The output from the program is shown here:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55
```

As this output shows, the for-each style **for** automatically cycles through an array in sequence from the lowest index to the highest.

Although the for-each **for** loop iterates until all elements in an array have been examined, it is possible to terminate the loop early by using a **break** statement. For example, this program sums only the first five elements of **nums**:

```
// Use break with a for-each style for.
class ForEach2 {
    public static void main(String[] args) {
        int sum = 0;
        int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        // use for to display and sum the values
        for(int x : nums) {
            System.out.println("Value is: " + x);
            sum += x;
            if(x == 5) break; // stop the loop when 5 is obtained
        }
        System.out.println("Summation of first 5 elements: " + sum);
    }
}
```

This is the output produced:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Summation of first 5 elements: 15
```

As is evident, the **for** loop stops after the fifth element has been obtained. The **break** statement can also be used with Java's other loops, and it is discussed in detail later in this chapter.

There is one important point to understand about the for-each style loop. Its iteration variable is “read-only” as it relates to the underlying array. An assignment to the iteration variable has no effect on the underlying array. In other words, you can't change the contents of the array by assigning the iteration variable a new value. For example, consider this program:

```

// The for-each loop is essentially read-only.
class NoChange {
    public static void main(String[] args) {
        int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        for(int x: nums) {
            System.out.print(x + " ");
            x = x * 10; // no effect on nums
        }

        System.out.println();

        for(int x : nums)
            System.out.print(x + " ");

        System.out.println();
    }
}

```

The first **for** loop increases the value of the iteration variable by a factor of 10. However, this assignment has no effect on the underlying array **nums**, as the second **for** loop illustrates. The output, shown here, proves this point:

```

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

```

Iterating Over Multidimensional Arrays

The enhanced version of the **for** also works on multidimensional arrays. Remember, however, that in Java, multidimensional arrays consist of *arrays of arrays*. (For example, a two-dimensional array is an array of one-dimensional arrays.) This is important when iterating over a multidimensional array, because each iteration obtains the *next array*, not an individual element. Furthermore, the iteration variable in the **for** loop must be compatible with the type of array being obtained. For example, in the case of a two-dimensional array, the iteration variable must be a reference to a one-dimensional array. In general, when using the for-each **for** to iterate over an array of N dimensions, the objects obtained will be arrays of $N-1$ dimensions. To understand the implications of this, consider

the following program. It uses nested **for** loops to obtain the elements of a two-dimensional array in row-order, from first to last.

```
// Use for-each style for on a two-dimensional array.
class ForEach3 {
    public static void main(String[] args) {
        int sum = 0;
        int[][] nums = new int[3][5];

        // give nums some values
        for(int i = 0; i < 3; i++)
            for(int j = 0; j < 5; j++)
                nums[i][j] = (i+1)*(j+1);

        // use for-each for to display and sum the values
        for(int[] x : nums) {
            for(int y : x) {
                System.out.println("Value is: " + y);
                sum += y;
            }
        }
        System.out.println("Summation: " + sum);
    }
}
```

The output from this program is shown here:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
Value is: 12
Value is: 15
Summation: 90
```

In the program, pay special attention to this line:

```
for(int[] x: nums) {
```

Notice how **x** is declared. It is a reference to a one-dimensional array of integers. This is necessary because each iteration of the **for** obtains the next *array* in **nums**, beginning with the array specified by **nums[0]**. The inner **for** loop then cycles through each of these arrays, displaying the values of each element.

Applying the Enhanced for

Since the for-each style **for** can only cycle through an array sequentially, from start to finish, you might think that its use is limited, but this is not true. A large number of algorithms require exactly this mechanism. One of the most common is searching. For example, the following program uses a **for** loop to search an unsorted array for a value. It stops if the value is found.

```

// Search an array using for-each style for.
class Search {
    public static void main(String[] args) {
        int[] nums = { 6, 8, 3, 7, 5, 6, 1, 4 };
        int val = 5;
        boolean found = false;

        // use for-each style for to search nums for val
        for(int x : nums) {
            if(x == val) {
                found = true;
                break;
            }
        }

        if(found)
            System.out.println("Value found!");
    }
}

```

The for-each style **for** is an excellent choice in this application because searching an unsorted array involves examining each element in sequence. (Of course, if the array were sorted, a binary search could be used, which would require a different style loop.) Other types of applications that benefit from for-each style loops include computing an average, finding the minimum or maximum of a set, looking for duplicates, and so on.

Although we have been using arrays in the examples in this chapter, the for-each style **for** is especially useful when operating on collections defined by the Collections Framework, which is described in Part II. More generally, the **for** can cycle through the elements of any collection of objects, as long as that collection satisfies a certain set of constraints, which are described in [Chapter 20](#).

Local Variable Type Inference in a for Loop

As explained in [Chapter 3](#), JDK 10 introduced a feature called *local variable type inference*, which allows the type of a local variable to be inferred from the type of its initializer. To use local variable type inference, the type of the variable is specified as **var** and the variable must be

initialized. Local variable type inference can be used in a **for** loop when declaring and initializing the loop control variable inside a traditional **for** loop, or when specifying the iteration variable in a for-each **for**. The following program shows an example of each case:

```
// Use type inference in a for loop.
class TypeInferenceInFor {
    public static void main(String[] args) {

        // Use type inference with the loop control variable.
        System.out.print("Values of x: ");
        for(var x = 2.5; x < 100.0; x = x * 2)
            System.out.print(x + " ");

        System.out.println();

        // Use type inference with the iteration variable.
        int[] nums = { 1, 2, 3, 4, 5, 6 };
        System.out.print("Values in nums array: ");
        for(var v : nums)
            System.out.print(v + " ");

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

The output is shown here:

```
Values of x: 2.5 5.0 10.0 20.0 40.0 80.0
Values in nums array: 1 2 3 4 5 6
```

In this example, loop control variable **x** in this line:

```
for(var x = 2.5; x < 100.0; x = x * 2)
```

is inferred to be type **double** because that is the type of its initializer.
Iteration variable **v** is this line:

```
for(var v : nums)
```

inferred to be of type **int** because that is the element type of the array **nums**.

One last point: Because a number of readers will be working in environments that predate JDK 10, local variable type inference will not be used by most of the **for** loops in the remainder of this edition of this book. You should, of course, consider it for new code that you write.

Nested Loops

Like all other programming languages, Java allows loops to be nested. That is, one loop may be inside another. For example, here is a program that nests **for** loops:

```
// Loops may be nested.
class Nested {
    public static void main(String[] args) {
        int i, j;

        for(i=0; i<10; i++) {
            for(j=i; j<10; j++)
                System.out.print(".");
            System.out.println();
        }
    }
}
```

The output produced by this program is shown here:

Jump Statements

Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control to another part of your program. Each is examined here.

NOTE In addition to the jump statements discussed here, Java supports one other way that you can change your program's flow of execution: through exception handling. Exception handling provides a structured method by which run-time errors can be trapped and handled by your program. It is supported by the keywords **try**, **catch**, **throw**, **throws**, and **finally**. In essence, the exception handling mechanism allows your program to perform a nonlocal branch. Since exception handling is a large topic, it is discussed in its own chapter, [Chapter 10](#).

Using **break**

In Java, the **break** statement has three uses. First, as you have seen, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop. Third, it can be used as a “civilized” form of `goto`. The last two uses are explained here.

Using **break** to Exit a Loop

By using **break**, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a **break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. Here is a simple example:

```
// Using break to exit a loop.
class BreakLoop {
    public static void main(String[] args) {
        for(int i=0; i<100; i++) {
            if(i == 10) break; // terminate loop if i is 10
            System.out.println("i: " + i);
        }
        System.out.println("Loop complete.");
    }
}
```

This program generates the following output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

As you can see, although the **for** loop is designed to run from 0 to 99, the **break** statement causes it to terminate early, when **i** equals 10.

The **break** statement can be used with any of Java's loops, including intentionally infinite loops. For example, here is the preceding program coded by use of a **while** loop. The output from this program is the same as just shown.

```
// Using break to exit a while loop.
class BreakLoop2 {
    public static void main(String[] args) {
        int i = 0;

        while(i < 100) {
            if(i == 10) break; // terminate loop if i is 10
            System.out.println("i: " + i);
            i++;
        }
        System.out.println("Loop complete.");
    }
}
```

When used inside a set of nested loops, the **break** statement will only break out of the innermost loop. For example:

```

// Using break with nested loops.
class BreakLoop3 {
    public static void main(String[] args) {
        for(int i=0; i<3; i++) {
            System.out.print("Pass " + i + ": ");
            for(int j=0; j<100; j++) {
                if(j == 10) break; // terminate loop if j is 10
                System.out.print(j + " ");
            }
            System.out.println();
        }
        System.out.println("Loops complete.");
    }
}

```

This program generates the following output:

```

Pass 0: 0 1 2 3 4 5 6 7 8 9
Pass 1: 0 1 2 3 4 5 6 7 8 9
Pass 2: 0 1 2 3 4 5 6 7 8 9
Loops complete.

```

As you can see, the **break** statement in the inner loop only causes termination of that loop. The outer loop is unaffected.

Here are two other points to remember about **break**. First, more than one **break** statement may appear in a loop. However, be careful. Too many **break** statements have the tendency to destructure your code. Second, the **break** that terminates a **switch** statement affects only that **switch** statement and not any enclosing loops.

REMEMBER **break** was not designed to provide the normal means by which a loop is terminated.

The loop's conditional expression serves this purpose. The **break** statement should be used to cancel a loop only when some sort of special situation occurs.

Using **break** as a Form of Goto

In addition to its uses with the **switch** statement and loops, the **break** statement can also be employed by itself to provide a “civilized” form of the goto statement. Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner. This usually makes goto-ridden code hard to understand and hard to maintain. It also

prohibits certain compiler optimizations. There are, however, a few places where the goto is a valuable and legitimate construct for flow control. For example, the goto can be useful when you are exiting from a deeply nested set of loops. To handle such situations, Java defines an expanded form of the **break** statement. By using this form of **break**, you can, for example, break out of one or more blocks of code. These blocks need not be part of a loop or a **switch**. They can be any block. Further, you can specify precisely where execution will resume, because this form of **break** works with a label. As you will see, **break** gives you the benefits of a goto without its problems.

The general form of the labeled **break** statement is shown here:

```
break label;
```

Most often, *label* is the name of a label that identifies a block of code. This can be a stand-alone block of code but it can also be a block that is the target of another statement. When this form of **break** executes, control is transferred out of the named block. The labeled block must enclose the **break** statement, but it does not need to be the immediately enclosing block. This means, for example, that you can use a labeled **break** statement to exit from a set of nested blocks. But you cannot use **break** to transfer control out of a block that does not enclose the **break** statement.

To name a block, put a label at the start of it. A *label* is any valid Java identifier followed by a colon. Once you have labeled a block, you can then use this label as the target of a **break** statement. Doing so causes execution to resume at the *end* of the labeled block. For example, the following program shows three nested blocks, each with its own label. The **break** statement causes execution to jump forward, past the end of the block labeled **second**, skipping the two **println()** statements.

```
// Using break as a civilized form of goto.
class Break {
    public static void main(String[] args) {
        boolean t = true;

        first: {
            second: {
                third: {
                    System.out.println("Before the break.");
                    if(t) break second; // break out of second block
                    System.out.println("This won't execute");
                }
                System.out.println("This won't execute");
            }
            System.out.println("This is after second block.");
        }
    }
}
```

Running this program generates the following output:

```
Before the break.
This is after second block.
```

One of the most common uses for a labeled **break** statement is to exit from nested loops. For example, in the following program, the outer loop executes only once:

```
// Using break to exit from nested loops
class BreakLoop4 {
    public static void main(String[] args) {
        outer: for(int i=0; i<3; i++) {
            System.out.print("Pass " + i + ": ");
            for(int j=0; j<100; j++) {
                if(j == 10) break outer; // exit both loops
                System.out.print(j + " ");
            }
        }
    }
}
```

```
        }
        System.out.println("This will not print");
    }
    System.out.println("Loops complete.");
}
}
```

This program generates the following output:

```
Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete.
```

As you can see, when the inner loop breaks to the outer loop, both loops have been terminated. Notice that this example labels the **for** statement, which has a block of code as its target.

Keep in mind that you cannot break to any label that is not defined for an enclosing block. For example, the following program is invalid and will not compile:

```
// This program contains an error.
class BreakErr {
    public static void main(String[] args) {

        one: for(int i=0; i<3; i++) {
            System.out.print("Pass " + i + ": ");
        }

        for(int j=0; j<100; j++) {
            if(j == 10) break one; // WRONG
            System.out.print(j + " ");
        }
    }
}
```

Since the loop labeled **one** does not enclose the **break** statement, it is not possible to transfer control out of that block.

Using continue

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The **continue** statement performs such an action. In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop. In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

Here is a sample program that uses **continue** to cause two numbers to be printed on each line:

```
// Demonstrate continue.
class Continue {
    public static void main(String[] args) {
        for(int i=0; i<10; i++) {
            System.out.print(i + " ");

            if (i%2 == 0) continue;
            System.out.println("");
        }
    }
}
```

This code uses the **%** operator to check if **i** is even. If it is, the loop continues without printing a newline. Here is the output from this program:

```
0 1
2 3
4 5
6 7
8 9
```

As with the **break** statement, **continue** may specify a label to describe which enclosing loop to continue. Here is a sample program that uses **continue** to print a triangular multiplication table for 0 through 9:

```
// Using continue with a label.
class ContinueLabel {
    public static void main(String[] args) {
outer: for (int i=0; i<10; i++) {
    for(int j=0; j<10; j++) {
        if(j > i) {
            System.out.println();
            continue outer;
        }
        System.out.print(" " + (i * j));
    }
}
System.out.println();
}
}
```

The **continue** statement in this example terminates the loop counting **j** and continues with the next iteration of the loop counting **i**. Here is the output of this program:

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

Good uses of **continue** are rare. One reason is that Java provides a rich set of loop statements that fit most applications. However, for those special circumstances in which early iteration is needed, the **continue** statement provides a structured way to accomplish it.

return

The last control statement is **return**. The **return** statement is used to explicitly return from a method. That is, it causes program control to

transfer back to the caller of the method. As such, it is categorized as a jump statement. Although a full discussion of **return** must wait until methods are discussed in [Chapter 6](#), a brief look at **return** is presented here.

At any time in a method, the **return** statement can be used to cause execution to branch back to the caller of the method. Thus, the **return** statement immediately terminates the method in which it is executed. The following example illustrates this point. Here, **return** causes execution to return to the Java run-time system, since it is the run-time system that calls **main()**:

```
// Demonstrate return.
class Return {
    public static void main(String[] args) {
        boolean t = true;

        System.out.println("Before the return.");
        if(t) return; // return to caller
        System.out.println("This won't execute.");
    }
}
```

The output from this program is shown here:

```
Before the return.
```

As you can see, the final **println()** statement is not executed. As soon as **return** is executed, control passes back to the caller.

One last point: In the preceding program, the **if(t)** statement is necessary. Without it, the Java compiler would flag an “unreachable code” error because the compiler would know that the last **println()** statement would never be executed. To prevent this error, the **if** statement is used here to trick the compiler for the sake of this demonstration.

CHAPTER

6

Introducing Classes

The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such, the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class.

Because the class is so fundamental to Java, this and the next few chapters will be devoted to it. Here, you will be introduced to the basic elements of a class and learn how a class can be used to create objects. You will also learn about methods, constructors, and the **this** keyword.

Class Fundamentals

Classes have been used since the beginning of this book. However, until now, only the most rudimentary form of a class has been shown. The classes created in the preceding chapters primarily exist simply to encapsulate the **main()** method, which has been used to demonstrate the basics of the Java syntax. As you will see, classes are substantially more powerful than the limited ones presented so far.

Perhaps the most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a *template* for an object, and an object is an *instance* of a class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.

The General Form of a Class

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both. As you will see, a class's code defines the interface to its data.

A class is declared by use of the **class** keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a **class** definition is shown here:

```
class classname {  
    type instance-variable1;  
    type instance-variable2;  
    // ...  
    type instance-variableN;  
  
    type methodname1(parameter-list) {  
        // body of method  
    }  
    type methodname2(parameter-list) {  
        // body of method  
    }  
    // ...  
    type methodnameN(parameter-list) {  
        // body of method  
    }  
}
```

The data, or variables, defined within a **class** are called *instance variables*. The code is contained within *methods*. Collectively, the methods and variables defined within a class are called *members* of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, as a general rule, it is the methods that determine how a class's data can be used.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. We will come back to this point shortly, but it is an important concept to learn early.

All methods have the same general form as **main()**, which we have been using thus far. However, most methods will not be specified as **static** or **public**. Notice that the general form of a class does not specify a **main()** method. Java classes do not need to have a **main()** method. You only specify one if that class is the starting point for your program. Further, some kinds of Java applications don't require a **main()** method at all.

A Simple Class

Let's begin our study of the class with a simple example. Here is a class called **Box** that defines three instance variables: **width**, **height**, and **depth**. Currently, **Box** does not contain any methods (but some will be added soon).

```
class Box {  
    double width;  
    double height;  
    double depth;  
}
```

As stated, a class defines a new type of data. In this case, the new data type is called **Box**. You will use this name to declare objects of type **Box**. It is important to remember that a class declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any objects of type **Box** to come into existence.

To actually create a **Box** object, you will use a statement like the following:

```
Box mybox = new Box(); // create a Box object called mybox
```

After this statement executes, **mybox** will refer to an instance of **Box**. Thus, it will have “physical” reality. For the moment, don't worry about the details of this statement.

As mentioned earlier, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every **Box** object will contain its own copies of the instance variables **width**, **height**, and **depth**. To access these variables, you will use the *dot* (.) operator. The dot operator links the name of the object with the name of an instance variable. For example, to assign the **width** variable of **mybox** the value 100, you would use the following statement:

```
mybox.width = 100;
```

This statement tells the compiler to assign the copy of **width** that is contained within the **mybox** object the value of 100. In general, you use the dot operator to access both the instance variables and the methods within an object. One other point: Although commonly referred to as the *dot operator*, the formal specification for Java categorizes the . as a separator. However, since the use of the term “dot operator” is widespread, it is used in this book.

Here is a complete program that uses the **Box** class:

```
/* A program that uses the Box class.  
   Call this file BoxDemo.java  
*/  
class Box {  
    double width;  
    double height;  
    double depth;  
}  
  
// This class declares an object of type Box.  
class BoxDemo {  
    public static void main(String[] args) {  
        Box mybox = new Box();  
        double vol;  
  
        // assign values to mybox's instance variables  
        mybox.width = 10;  
        mybox.height = 20;  
        mybox.depth = 15;  
  
        // compute volume of box  
        vol = mybox.width * mybox.height * mybox.depth;  
  
        System.out.println("Volume is " + vol);  
    }  
}
```

You should call the file that contains this program **BoxDemo.java**, because the **main()** method is in the class called **BoxDemo**, not the class called **Box**. When you compile this program, you will find that two **.class** files have been created, one for **Box** and one for **BoxDemo**. The Java compiler automatically puts each class into its own **.class** file. It is not necessary for both the **Box** and the **BoxDemo** class to actually be in the same source file. You could put each class in its own file, called **Box.java** and **BoxDemo.java**, respectively.

To run this program, you must execute **BoxDemo.class**. When you do, you will see the following output:

```
Volume is 3000.0
```

As stated earlier, each object has its own copies of the instance variables. This means that if you have two **Box** objects, each has its own copy of **depth**, **width**, and **height**. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another. For example, the following program declares two **Box** objects:

```
// This program declares two Box objects.

class Box {
    double width;
    double height;
    double depth;
}

class BoxDemo2 {
    public static void main(String[] args) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* assign different values to mybox2's
           instance variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // compute volume of first box
        vol = mybox1.width * mybox1.height * mybox1.depth;
        System.out.println("Volume is " + vol);

        // compute volume of second box
        vol = mybox2.width * mybox2.height * mybox2.depth;
        System.out.println("Volume is " + vol);
    }
}
```

The output produced by this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

As you can see, **mybox1**'s data is completely separate from the data contained in **mybox2**.

Declaring Objects

As just explained, when you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator. The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, essentially, the address in memory of the object allocated by **new**. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let's look at the details of this procedure.

In the preceding sample programs, a line similar to the following is used to declare an object of type **Box**:

```
Box mybox = new Box();
```

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

The first line declares **mybox** as a reference to an object of type **Box**. At this point, **mybox** does not yet refer to an actual object. The next line allocates an object and assigns a reference to it to **mybox**. After the second line executes, you can use **mybox** as if it were a **Box** object. But in reality,

`mybox` simply holds, in essence, the memory address of the actual **Box** object. The effect of these two lines of code is depicted in [Figure 6-1](#).

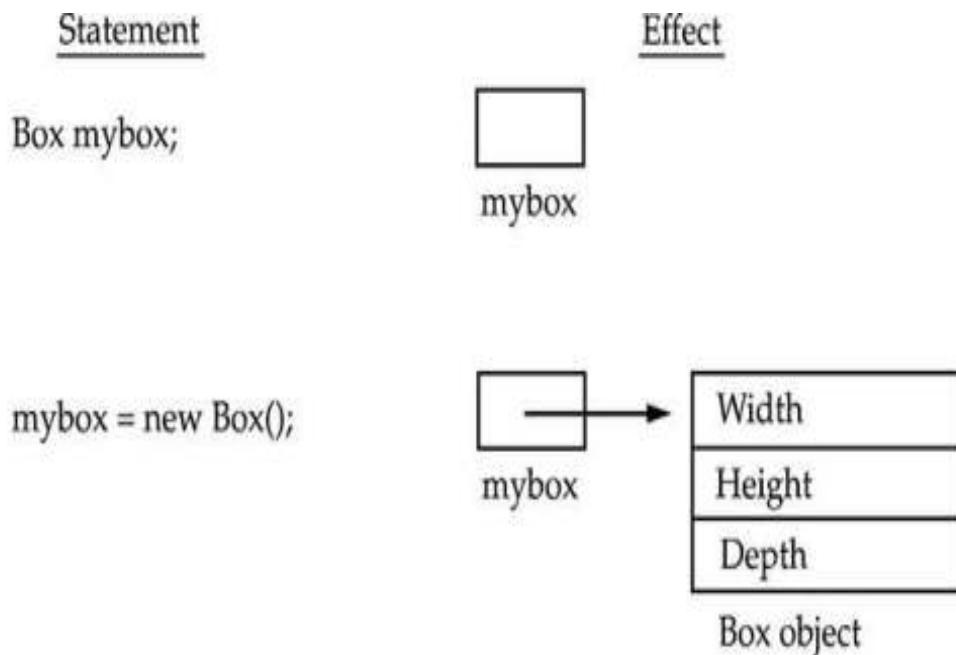


Figure 6-1 Declaring an object of type **Box**

A Closer Look at `new`

As just explained, the `new` operator dynamically allocates memory for an object. In the context of an assignment, it has this general form:

class-var = new *classname* ();

Here, *class-var* is a variable of the class type being created. The *classname* is the name of the class that is being instantiated. The class name followed by parentheses specifies the *constructor* for the class. A constructor defines what occurs when an object of a class is created. Constructors are an important part of all classes and have many significant attributes. Most real-world classes explicitly define their own constructors within their class definition. However, if no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with **Box**. For now, we will use the default constructor. Soon, you will see how to define your own constructors.

At this point, you might be wondering why you do not need to use **new** for such things as integers or characters. The answer is that Java's primitive types are not implemented as objects. Rather, they are implemented as "normal" variables. This is done in the interest of efficiency. As you will see, objects have many features and attributes that require Java to treat them differently than it treats the primitive types. By not applying the same overhead to the primitive types that applies to objects, Java can implement the primitive types more efficiently. Later, you will see object versions of the primitive types that are available for your use in those situations in which complete objects of these types are needed.

It is important to understand that **new** allocates memory for an object during run time. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program. However, since memory is finite, it is possible that **new** will not be able to allocate memory for an object because insufficient memory exists. If this happens, a run-time exception will occur. (You will learn how to handle exceptions in [Chapter 10](#).) For the sample programs in this book, you won't need to worry about running out of memory, but you will need to consider this possibility in real-world programs that you write.

Let's once again review the distinction between a class and an object. A class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class. Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.) It is important to keep this distinction clearly in mind.

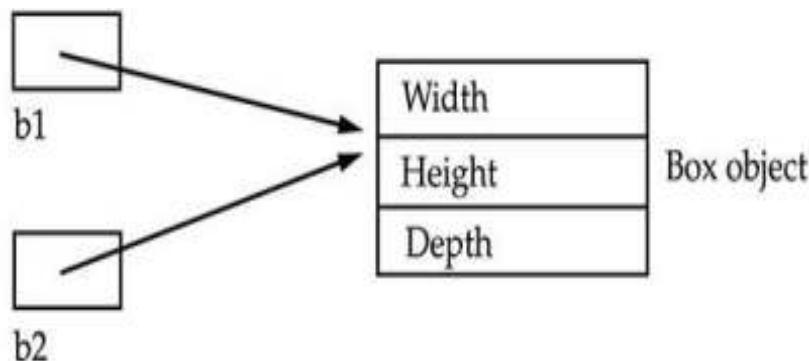
Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

```
Box b1 = new Box();
Box b2 = b1;
```

You might think that **b2** is being assigned a reference to a copy of the object referred to by **b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the *same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.

This situation is depicted here:



Although **b1** and **b2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**. For example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, **b1** has been set to **null**, but **b2** still points to the original object.

REMEMBER When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

Introducing Methods

As mentioned at the beginning of this chapter, classes usually consist of two things: instance variables and methods. The topic of methods is a large one because Java gives them so much power and flexibility. In fact, much

of the next chapter is devoted to methods. However, there are some fundamentals that you need to learn now so that you can begin to add methods to your classes.

This is the general form of a method:

```
type name(parameter-list) {  
    // body of method  
}
```

Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**. The name of the method is specified by *name*. This can be any legal identifier other than those already used by other items within the current scope. The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.

Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

```
return value;
```

Here, *value* is the value returned.

In the next few sections, you will see how to create various types of methods, including those that take parameters and those that return values.

Adding a Method to the Box Class

Although it is perfectly fine to create a class that contains only data, it rarely happens. Most of the time, you will use methods to access the instance variables defined by the class. In fact, methods define the interface to most classes. This allows the class implementor to hide the specific layout of internal data structures behind cleaner method abstractions. In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself.

Let's begin by adding a method to the **Box** class. It may have occurred to you while looking at the preceding programs that the computation of a box's volume was something that was best handled by the **Box** class rather than the **BoxDemo** class. After all, since the volume of a box is dependent upon the size of the box, it makes sense to have the **Box** class compute it. To do this, you must add a method to **Box**, as shown here:

```
// This program includes a method inside the box class.

class Box {
    double width;
    double height;
    double depth;

    // display volume of a box
    void volume() {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);
    }
}

class BoxDemo3 {
    public static void main(String[] args) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
```

```
/* assign different values to mybox2's
   instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;

// display volume of first box
mybox1.volume();

// display volume of second box
mybox2.volume();
}
}
```

This program generates the following output, which is the same as the previous version.

```
Volume is 3000.0
Volume is 162.0
```

Look closely at the following two lines of code:

```
mybox1.volume();
mybox2.volume();
```

The first line here invokes the **volume()** method on **mybox1**. That is, it calls **volume()** relative to the **mybox1** object, using the object's name followed by the dot operator. Thus, the call to **mybox1.volume()** displays the volume of the box defined by **mybox1**, and the call to **mybox2.volume()** displays the volume of the box defined by **mybox2**. Each time **volume()** is invoked, it displays the volume for the specified box.

If you are unfamiliar with the concept of calling a method, the following discussion will help clear things up. When **mybox1.volume()** is executed, the Java run-time system transfers control to the code defined inside **volume()**. After the statements inside **volume()** have executed, control is returned to the calling routine, and execution resumes with the line of code following the call. In the most general sense, a method is Java's way of implementing subroutines.

There is something very important to notice inside the **volume()** method: the instance variables **width**, **height**, and **depth** are referred to directly, without preceding them with an object name or the dot operator. When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator. This is easy to understand if you think about it. A method is always invoked relative to some object of its class. Once this invocation has occurred, the object is known. Thus, within a method, there is no need to specify the object a second time. This means that **width**, **height**, and **depth** inside **volume()** implicitly refer to the copies of those variables found in the object that invokes **volume()**.

Let's review: When an instance variable is accessed by code that is not part of the class in which that instance variable is defined, it must be done through an object, by use of the dot operator. However, when an instance variable is accessed by code that is part of the same class as the instance variable, that variable can be referred to directly. The same thing applies to methods.

Returning a Value

While the implementation of **volume()** does move the computation of a box's volume inside the **Box** class where it belongs, this is not the best way to do it. For example, what if another part of your program wanted to know the volume of a box but not display its value? A better way to implement **volume()** is to have it compute the volume of the box and return the result to the caller. The following example, an improved version of the preceding program, does just that:

```
// Now, volume() returns the volume of a box.

class Box {
    double width;
    double height;
    double depth;

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo4 {
    public static void main(String[] args) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* assign different values to mybox2's
           instance variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

As you can see, when **volume()** is called, it is put on the right side of an assignment statement. On the left is a variable, in this case **vol**, that will receive the value returned by **volume()**. Thus, after

```
vol = mybox1.volume();
```

executes, the value of **mybox1.volume()** is 3000 and this value then is stored in **vol**.

There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean**, you could not return an integer.
- The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method.

One more point: The preceding program can be written a bit more efficiently because there is actually no need for the **vol** variable. The call to **volume()** could have been used in the **println()** statement directly, as shown here:

```
System.out.println("Volume is" + mybox1.volume());
```

In this case, when **println()** is executed, **mybox1.volume()** will be called automatically and its value will be passed to **println()**.

Adding a Method That Takes Parameters

While some methods don't need parameters, most do. Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations. To illustrate this point, let's use a very simple example. Here is a method that returns the square of the number 10:

```
int square()
{
    return 10 * 10;
}
```

While this method does, indeed, return the value of 10 squared, its use is very limited. However, if you modify the method so that it takes a parameter, as shown next, then you can make **square()** much more useful.

```
int square(int i)
{
    return i * i;
}
```

Now, **square()** will return the square of whatever value it is called with. That is, **square()** is now a general-purpose method that can compute the square of any integer value, rather than just 10.

Here is an example:

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

In the first call to **square()**, the value 5 will be passed into parameter **i**. In the second call, **i** will receive the value 9. The third invocation passes the value of **y**, which is 2 in this example. As these examples show, **square()** is able to return the square of whatever data it is passed.

It is important to keep the two terms *parameter* and *argument* straight. A *parameter* is a variable defined by a method that receives a value when the method is called. For example, in **square()**, **i** is a parameter. An *argument* is a value that is passed to a method when it is invoked. For example, **square(100)** passes 100 as an argument. Inside **square()**, the parameter **i** receives that value.

You can use a parameterized method to improve the **Box** class. In the preceding examples, the dimensions of each box had to be set separately by use of a sequence of statements, such as:

```
mybox1.width = 10;  
mybox1.height = 20;  
mybox1.depth = 15;
```

While this code works, it is troubling for two reasons. First, it is clumsy and error prone. For example, it would be easy to forget to set a dimension. Second, in well-designed Java programs, instance variables should be accessed only through methods defined by their class. In the future, you can change the behavior of a method, but you can't change the behavior of an exposed instance variable.

Thus, a better approach to setting the dimensions of a box is to create a method that takes the dimensions of a box in its parameters and sets each instance variable appropriately. This concept is implemented by the following program:

```
// This program uses a parameterized method.  
  
class Box {  
    double width;  
    double height;  
    double depth;  
  
    // compute and return volume  
    double volume() {  
        return width * height * depth;  
    }  
  
    // sets dimensions of box  
    void setDim(double w, double h, double d) {  
        width = w;  
        height = h;
```

```

        depth = d;
    }
}

class BoxDemo5 {
    public static void main(String[] args) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // initialize each box
        mybox1.setDim(10, 20, 15);
        mybox2.setDim(3, 6, 9);

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}

```

As you can see, the **setDim()** method is used to set the dimensions of each box. For example, when

```
mybox1.setDim(10, 20, 15);
```

is executed, 10 is copied into parameter **w**, 20 is copied into **h**, and 15 is copied into **d**. Inside **setDim()** the values of **w**, **h**, and **d** are then assigned to **width**, **height**, and **depth**, respectively.

For many readers, the concepts presented in the preceding sections will be familiar. However, if such things as method calls, arguments, and parameters are new to you, then you might want to take some time to experiment before moving on. The concepts of the method invocation, parameters, and return values are fundamental to Java programming.

Constructors

It can be tedious to initialize all of the variables in a class each time an instance is created. Even when you add convenience methods like `setDim()`, it would be simpler and more concise to have all of the setup done at the time the object is first created. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.

A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called when the object is created, before the `new` operator completes. Constructors look a little strange because they have no return type, not even `void`. This is because the implicit return type of a class's constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

You can rework the **Box** example so that the dimensions of a box are automatically initialized when an object is constructed. To do so, replace `setDim()` with a constructor. Let's begin by defining a simple constructor that sets the dimensions of each box to the same values. This version is shown here:

```
/* Here, Box uses a constructor to initialize the
   dimensions of a box.
*/
class Box {
    double width;
    double height;
    double depth;

    // This is the constructor for Box.
    Box() {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo6 {
    public static void main(String[] args) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

When this program is run, it generates the following results:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

As you can see, both **mybox1** and **mybox2** were initialized by the **Box()** constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both **mybox1** and **mybox2** will have the same volume. The **println()** statement inside **Box()** is for the sake of illustration only. Most constructors will not display anything. They will simply initialize an object.

Before moving on, let's reexamine the **new** operator. As you know, when you allocate an object, you use the following general form:

```
class-var = new classname ( );
```

Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line

```
Box mybox1 = new Box();
```

new Box() is calling the **Box()** constructor. When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. This is why the preceding line of code worked in earlier versions of **Box** that did not define a constructor. When using the default constructor, all non-initialized instance variables will have their default values, which are zero, **null**, and **false**, for numeric types, reference types, and **boolean**, respectively. The default constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones. Once you define your own constructor, the default constructor is no longer used.

Parameterized Constructors

While the **Box()** constructor in the preceding example does initialize a **Box** object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct **Box** objects of various dimensions. The easy solution is to add parameters to the constructor. As you can probably guess,

this makes it much more useful. For example, the following version of **Box** defines a parameterized constructor that sets the dimensions of a box as specified by those parameters. Pay special attention to how **Box** objects are created.

```
/* Here, Box uses a parameterized constructor to
   initialize the dimensions of a box.
*/
class Box {
    double width;
    double height;
    double depth;

    // This is the constructor for Box.
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // compute and return volume
    double volume() {
```

```

        return width * height * depth;
    }
}

class BoxDemo7 {
    public static void main(String[] args) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box(3, 6, 9);

        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}

```

The output from this program is shown here:

```

Volume is 3000.0
Volume is 162.0

```

As you can see, each object is initialized as specified in the parameters to its constructor. For example, in the following line,

```
Box mybox1 = new Box(10, 20, 15);
```

the values 10, 20, and 15 are passed to the **Box()** constructor when **new** creates the object. Thus, **mybox1**'s copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15, respectively.

The **this** Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any

method to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class's type is permitted.

To better understand what **this** refers to, consider the following version of **Box()**:

```
// A redundant use of this.  
Box(double w, double h, double d) {  
    this.width = w;  
    this.height = h;  
    this.depth = d;  
}
```

This version of **Box()** operates exactly like the earlier version. The use of **this** is redundant but perfectly correct. Inside **Box()**, **this** will always refer to the invoking object. While it is redundant in this case, **this** is useful in other contexts, one of which is explained in the next section.

Instance Variable Hiding

As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes. Interestingly, you can have local variables, including formal parameters to methods, that overlap with the names of the class's instance variables. However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable. This is why **width**, **height**, and **depth** were not used as the names of the parameters to the **Box()** constructor inside the **Box** class. If they had been, then **width**, for example, would have referred to the formal parameter, hiding the instance variable **width**. While it is usually easier to simply use different names, there is another way around this situation. Because **this** lets you refer directly to the object, you can use it to resolve any namespace collisions that might occur between instance variables and local variables. For example, here is another version of **Box()**, which uses **width**, **height**, and **depth** for parameter names and then uses **this** to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.  
Box(double width, double height, double depth) {  
    this.width = width;  
    this.height = height;  
    this.depth = depth;  
}
```

A word of caution: The use of **this** in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables. Of course, other programmers believe the contrary—that it is a good convention to use the same names for clarity—and use **this** to overcome the instance variable hiding. It is a matter of taste which approach you adopt.

Garbage Collection

Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as traditional C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection*. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no need to explicitly destroy objects. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

A Stack Class

While the **Box** class is useful to illustrate the essential elements of a class, it is of little practical value. To show the real power of classes, this chapter will conclude with a more sophisticated example. As you recall from the discussion of object-oriented programming (OOP) presented in [Chapter 2](#),

one of OOP’s most important benefits is the encapsulation of data and the code that manipulates that data. As you have seen, the class is the mechanism by which encapsulation is achieved in Java. By creating a class, you are creating a new data type that defines both the nature of the data being manipulated and the routines used to manipulate it. Further, the methods define a consistent and controlled interface to the class’s data. Thus, you can use the class through its methods without having to worry about the details of its implementation or how the data is actually managed within the class. In a sense, a class is like a “data engine.” No knowledge of what goes on inside the engine is required to use the engine through its controls. In fact, since the details are hidden, its inner workings can be changed as needed. As long as your code uses the class through its methods, internal details can change without causing side effects outside the class.

To see a practical application of the preceding discussion, let’s develop one of the archetypal examples of encapsulation: the stack. A *stack* stores data using first-in, last-out ordering. That is, a stack is like a stack of plates on a table—the first plate put down on the table is the last plate to be used. Stacks are controlled through two operations traditionally called *push* and *pop*. To put an item on top of the stack, you will use push. To take an item off the stack, you will use pop. As you will see, it is easy to encapsulate the entire stack mechanism.

Here is a class called **Stack** that implements a stack for up to ten integers:

```
// This class defines an integer stack that can hold 10 values
class Stack {
    int[] stck = new int[10];
    int tos;

    // Initialize top-of-stack
    Stack() {
        tos = -1;
    }

    // Push an item onto the stack
    void push(int item) {
        if(tos==9)
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }

    // Pop an item from the stack
    int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}
```

As you can see, the **Stack** class defines two data items, two methods, and a constructor. The stack of integers is held by the array **stck**. This array is indexed by the variable **tos**, which always contains the index of the top of the stack. The **Stack()** constructor initializes **tos** to -1 , which indicates an empty stack. The method **push()** puts an item on the stack. To retrieve an item, call **pop()**. Since access to the stack is through **push()** and **pop()**, the fact that the stack is held in an array is actually not relevant to using the stack. For example, the stack could be held in a more complicated data structure, such as a linked list, yet the interface defined by **push()** and **pop()** would remain the same.

The class **TestStack**, shown here, demonstrates the **Stack** class. It creates two integer stacks, pushes some values onto each, and then pops them off.

```
class TestStack {  
    public static void main(String[] args) {  
        Stack mystack1 = new Stack();  
        Stack mystack2 = new Stack();  
  
        // push some numbers onto the stack  
        for(int i=0; i<10; i++) mystack1.push(i);  
        for(int i=10; i<20; i++) mystack2.push(i);  
  
        // pop those numbers off the stack  
        System.out.println("Stack in mystack1:");  
        for(int i=0; i<10; i++)  
            System.out.println(mystack1.pop());  
  
        System.out.println("Stack in mystack2:");  
        for(int i=0; i<10; i++)  
            System.out.println(mystack2.pop());  
    }  
}
```

This program generates the following output:

```
Stack in mystack1:  
9  
8  
7  
6  
5  
4  
3  
2  
1  
0
```

```
Stack in mystack2:  
19  
18  
17  
16  
15  
14  
13  
12  
11  
10
```

As you can see, the contents of each stack are separate.

One last point about the **Stack** class. As it is currently implemented, it is possible for the array that holds the stack, **stck**, to be altered by code outside of the **Stack** class. This leaves **Stack** open to misuse or mischief. In the next chapter, you will see how to remedy this situation.

CHAPTER

7

A Closer Look at Methods and Classes

This chapter continues the discussion of methods and classes begun in the preceding chapter. It examines several topics relating to methods, including overloading, parameter passing, and recursion. The chapter then returns to the class, discussing access control, the use of the keyword **static**, and one of Java's most important built-in classes: **String**.

Overloading Methods

In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as *method overloading*. Method overloading is one of the ways that Java supports polymorphism. If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. But as you will see, method overloading is one of Java's most exciting and useful features.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an

overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }

    // Overload test for one integer parameter.
    void test(int a) {
        System.out.println("a: " + a);
    }

    // Overload test for two integer parameters.
    void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
    }

    // Overload test for a double parameter
    double test(double a) {
        System.out.println("double a: " + a);
        return a*a;
    }
}

class Overload {
    public static void main(String[] args) {
        OverloadDemo ob = new OverloadDemo();
        double result;

        // call all versions of test()
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25): " + result);
    }
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

As you can see, **test()** is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one **double** parameter. The fact that the fourth version of **test()** also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters. However, this match need not always be exact. In some cases, Java's automatic type conversions can play a role in overload resolution. For example, consider the following program:

```
// Automatic type conversions apply to overloading.
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }

    // Overload test for two integer parameters.
    void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
    }
}
```

```
// Overload test for a double parameter
void test(double a) {
    System.out.println("Inside test(double) a: " + a);
}
}

class Overload {
    public static void main(String[] args) {
        OverloadDemo ob = new OverloadDemo();
        int i = 88;

        ob.test();
        ob.test(10, 20);

        ob.test(i); // this will invoke test(double)
        ob.test(123.2); // this will invoke test(double)
    }
}
```

This program generates the following output:

```
No parameters
a and b: 10 20
Inside test(double) a: 88.0
Inside test(double) a: 123.2
```

As you can see, this version of **OverloadDemo** does not define **test(int)**. Therefore, when **test()** is called with an integer argument inside **Overload**, no matching method is found. However, Java can automatically convert an integer into a **double**, and this conversion can be used to resolve the call. Therefore, after **test(int)** is not found, Java elevates **i** to **double** and then calls **test(double)**. Of course, if **test(int)** had been defined, it would have been called instead. Java will employ its automatic type conversions only if no exact match is found.

Method overloading supports polymorphism because it is one way that Java implements the “one interface, multiple methods” paradigm. To understand how, consider the following. In languages that do not support method overloading, each method must be given a unique name. However, frequently you will want to implement essentially the same method for different types of data. Consider the absolute value function. In languages

that do not support overloading, there are usually three or more versions of this function, each with a slightly different name. For instance, in C, the function **abs()** returns the absolute value of an integer, **labs()** returns the absolute value of a long integer, and **fabs()** returns the absolute value of a floating-point value. Since C does not support overloading, each function has its own name, even though all three functions do essentially the same thing. This makes the situation more complex, conceptually, than it actually is. Although the underlying concept of each function is the same, you still have three names to remember. This situation does not occur in Java, because each absolute value method can use the same name. Indeed, Java's standard class library includes an absolute value method, called **abs()**. This method is overloaded by Java's **Math** class to handle all numeric types. Java determines which version of **abs()** to call based upon the type of argument.

The value of overloading is that it allows related methods to be accessed by use of a common name. Thus, the name **abs** represents the *general action* that is being performed. It is left to the compiler to choose the right *specific* version for a particular circumstance. You, the programmer, need only remember the general operation being performed. Through the application of polymorphism, several names have been reduced to one. Although this example is fairly simple, if you expand the concept, you can see how overloading can help you manage greater complexity.

When you overload a method, each version of that method can perform any activity you desire. There is no rule stating that overloaded methods must relate to one another. However, from a stylistic point of view, method overloading implies a relationship. Thus, while you can use the same name to overload unrelated methods, you should not. For example, you could use the name **sqr** to create methods that return the *square* of an integer and the *square root* of a floating-point value. But these two operations are fundamentally different. Applying method overloading in this manner defeats its original purpose. In practice, you should only overload closely related operations.

Overloading Constructors

In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create,

overloaded constructors will be the norm, not the exception. To understand why, let's return to the **Box** class developed in the preceding chapter. Following is the latest version of **Box**:

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    // This is the constructor for Box.  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
  
    // compute and return volume  
    double volume() {  
        return width * height * depth;  
    }  
}
```

As you can see, the **Box()** constructor requires three parameters. This means that all declarations of **Box** objects must pass three arguments to the **Box()** constructor. For example, the following statement is currently invalid:

```
Box ob = new Box();
```

Since **Box()** requires three arguments, it's an error to call it without them. This raises some important questions. What if you simply wanted a box and did not care (or know) what its initial dimensions were? Or, what if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions? As the **Box** class is currently written, these other options are not available to you.

Fortunately, the solution to these problems is quite easy: simply overload the **Box** constructor so that it handles the situations just described. Here is a program that contains an improved version of **Box** that does just that:

```
/* Here, Box defines three constructors to initialize
   the dimensions of a box various ways.
*/
class Box {
    double width;
    double height;
    double depth;

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class OverloadCons {
    public static void main(String[] args) {
        // create boxes using the various constructors
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);

        double vol;
```

```
// get volume of cube  
vol = mycube.volume();  
System.out.println("Volume of mycube is " + vol);  
}  
}  
// get volume of second box  
vol = mybox2.volume();  
The output produced by this program is shown here:  
System.out.println("Volume of mybox2 is " + vol);  
  
Volume of mybox1 is 3000.0  
Volume of mybox2 is -1.0  
Volume of mycube is 343.0
```

As you can see, the proper overloaded constructor is called based upon the arguments specified when **new** is executed.

Using Objects as Parameters

So far, we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods. For example, consider the following short program:

```
// Objects may be passed to methods.
class Test {
    int a, b;

    Test(int i, int j) {
        a = i;
        b = j;
    }

    // return true if o is equal to the invoking object
    boolean equalTo(Test o) {
        if(o.a == a && o.b == b) return true;
        else return false;
    }
}

class PassOb {
    public static void main(String[] args) {
        Test ob1 = new Test(100, 22);
        Test ob2 = new Test(100, 22);
        Test ob3 = new Test(-1, -1);

        System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
        System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
    }
}
```

This program generates the following output:

```
ob1 == ob2: true
ob1 == ob3: false
```

As you can see, the **equalTo()** method inside **Test** compares two objects for equality and returns the result. That is, it compares the invoking object with the one that it is passed. If they contain the same values, then the method returns **true**. Otherwise, it returns **false**. Notice that the parameter **o** in **equalTo()** specifies **Test** as its type. Although **Test** is a class type created by the program, it is used in just the same way as Java's built-in types.

One of the most common uses of object parameters involves constructors. Frequently, you will want to construct a new object so that it is initially the same as some existing object. To do this, you must define a constructor that takes an object of its class as a parameter. For example, the following version of **Box** allows one object to initialize another:

```
// Here, Box allows one object to initialize another.

class Box {
    double width;
    double height;
    double depth;

    // Notice this constructor. It takes an object of type Box.
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class OverloadCons2 {
    public static void main(String[] args) {
        // create boxes using the various constructors
    }
}
```

```
Box mybox2 = new Box();
Box mycube = new Box(7);

Box myclone = new Box(mybox1); // create copy of mybox1

double vol;

// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);

// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);

// get volume of cube
vol = mycube.volume();
System.out.println("Volume of cube is " + vol);

// get volume of clone
vol = myclone.volume();
System.out.println("Volume of clone is " + vol);
}
}
```

As you will see when you begin to create your own classes, providing many forms of constructors is usually required to allow objects to be constructed in a convenient and efficient manner.

A Closer Look at Argument Passing

In general, there are two ways that a computer language can pass an argument to a subroutine. The first way is *call-by-value*. This approach copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument. The second way an argument can be passed is *call-by-reference*. In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This

means that changes made to the parameter will affect the argument used to call the subroutine. As you will see, although Java uses call-by-value to pass all arguments, the precise effect differs between whether a primitive type or a reference type is passed.

When you pass a primitive type to a method, it is passed by value. Thus, a copy of the argument is made, and what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

```
// Primitive types are passed by value.
class Test {
    void meth(int i, int j) {
        i *= 2;
        j /= 2;
    }
}

class CallByValue {
    public static void main(String[] args) {
        Test ob = new Test();

        int a = 15, b = 20;

        System.out.println("a and b before call: " +
                           a + " " + b);

        ob.meth(a, b);

        System.out.println("a and b after call: " +
                           a + " " + b);
    }
}
```

The output from this program is shown here:

```
a and b before call: 15 20
a and b after call: 15 20
```

As you can see, the operations that occur inside **meth()** have no effect on the values of **a** and **b** used in the call; their values here did not change to 30

and 10.

When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference. Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects act as if they are passed to methods by use of call-by-reference. Changes to the object inside the method *do* affect the object used as an argument. For example, consider the following program:

```
// Objects are passed through their references.

class Test {
    int a, b;

    Test(int i, int j) {
        a = i;
        b = j;
    }

    // pass an object
    void meth(Test o) {
        o.a *= 2;
        o.b /= 2;
    }
}

class PassObjRef {
    public static void main(String[] args) {
        Test ob = new Test(15, 20);
```

```
System.out.println("ob.a and ob.b before call: " +
    ob.a + " " + ob.b);

ob.meth(ob);

System.out.println("ob.a and ob.b after call: " +
    ob.a + " " + ob.b);
}
}
```

This program generates the following output:

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```

As you can see, in this case, the actions inside **meth()** have affected the object used as an argument.

REMEMBER When an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does.

Returning Objects

A method can return any type of data, including class types that you create. For example, in the following program, the **incrByTen()** method returns an object in which the value of **a** is ten greater than it is in the invoking object.

```

// Returning an object.
class Test {
    int a;

    Test(int i) {
        a = i;
    }

    Test incrByTen() {
        Test temp = new Test(a+10);
        return temp;
    }
}

class RetOb {
    public static void main(String[] args) {
        Test ob1 = new Test(2);
        Test ob2;

        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);

        ob2 = ob2.incrByTen();
        System.out.println("ob2.a after second increase: "
                           + ob2.a);
    }
}

```

The output generated by this program is shown here:

```

ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22

```

As you can see, each time **incrByTen()** is invoked, a new object is created, and a reference to it is returned to the calling routine.

The preceding program makes another important point: Since all objects are dynamically allocated using **new**, you don't need to worry about an object going out of scope because the method in which it was created

terminates. The object will continue to exist as long as there is a reference to it somewhere in your program. When there are no references to it, the object will be reclaimed the next time garbage collection takes place.

Recursion

Java supports *recursion*. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be *recursive*.

The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N . For example, 3 factorial is $1 \times 2 \times 3$, or 6. Here is how a factorial can be computed by use of a recursive method:

```
// A simple example of recursion.
class Factorial {
    // this is a recursive method
    int fact(int n) {
        int result;

        if(n==1) return 1;
        result = fact(n-1) * n;
        return result;
    }
}

class Recursion {
    public static void main(String[] args) {
        Factorial f = new Factorial();

        System.out.println("Factorial of 3 is " + f.fact(3));
        System.out.println("Factorial of 4 is " + f.fact(4));
        System.out.println("Factorial of 5 is " + f.fact(5));
    }
}
```

The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

If you are unfamiliar with recursive methods, then the operation of **fact()** may seem a bit confusing. Here is how it works. When **fact()** is called with an argument of 1, the function returns 1; otherwise, it returns the product of **fact(n-1)*n**. To evaluate this expression, **fact()** is called with **n-1**. This process repeats until **n** equals 1 and the calls to the method begin returning.

To better understand how the **fact()** method works, let's go through a short example. When you compute the factorial of 3, the first call to **fact()** will cause a second call to be made with an argument of 2. This invocation will cause **fact()** to be called a third time with an argument of 1. This call will return 1, which is then multiplied by 2 (the value of **n** in the second invocation). This result (which is 2) is then returned to the original invocation of **fact()** and multiplied by 3 (the original value of **n**). This yields the answer, 6. You might find it interesting to insert **println()** statements into **fact()**, which will show at what level each call is and what the intermediate answers are.

When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method. Recursive methods could be said to “telescope” out and back.

Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional method calls. A large number of recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted. If this occurs, the Java run-time system will cause an exception. However, this is typically not an issue unless a recursive routine runs wild.

The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives. For example, the QuickSort sorting algorithm is quite

difficult to implement in an iterative way. Also, some types of AI-related algorithms are most easily implemented using recursive solutions.

When writing recursive methods, you must have an **if** statement somewhere to force the method to return without the recursive call being executed. If you don't do this, once you call the method, it will never return. This is a very common error in working with recursion. Use **println()** statements liberally during development so that you can watch what is going on and abort execution if you see that you have made a mistake.

Here is one more example of recursion. The recursive method **printArray()** prints the first **i** elements in the array **values**.

```
// Another example that uses recursion.

class RecTest {
    int[] values;

    RecTest(int i) {
        values = new int[i];
    }

    // display array -- recursively
    void printArray(int i) {
        if(i==0) return;
        else printArray(i-1);
        System.out.println("[" + (i-1) + "] " + values[i-1]);
    }
}

class Recursion2 {
    public static void main(String[] args) {
        RecTest ob = new RecTest(10);
        int i;

        for(i=0; i<10; i++) ob.values[i] = i;
        ob.printArray(10);
    }
}
```

This program generates the following output:

```
[0] 0  
[1] 1  
[2] 2  
[3] 3  
[4] 4  
[5] 5  
[6] 6  
[7] 7  
[8] 8  
[9] 9
```

Introducing Access Control

As you know, encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: *access control*. Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse. For example, allowing access to data only through a well-defined set of methods, you can prevent the misuse of that data. Thus, when correctly implemented, a class creates a “black box” that may be used, but the inner workings of which are not open to tampering. However, the classes that were presented earlier do not completely meet this goal. For example, consider the **Stack** class shown at the end of [Chapter 6](#). While it is true that the methods **push()** and **pop()** do provide a controlled interface to the stack, this interface is not enforced. That is, it is possible for another part of the program to bypass these methods and access the stack directly. Of course, in the wrong hands, this could lead to trouble. In this section, you will be introduced to the mechanism by which you can precisely control access to the various members of a class.

How a member can be accessed is determined by the *access modifier* attached to its declaration. Java supplies a rich set of access modifiers. Some aspects of access control are related mostly to inheritance or packages. (A *package* is, essentially, a grouping of classes.) These parts of Java’s access control mechanism will be discussed in subsequent chapters. Here, let’s begin by examining access control as it applies to a single class.

Once you understand the fundamentals of access control, the rest will be easy.

NOTE The modules feature added by JDK 9 can also impact accessibility. Modules are described in [Chapter 16](#).

Java's access modifiers are **public**, **private**, and **protected**. Java also defines a default access level. **protected** applies only when inheritance is involved. The other access modifiers are described next.

Let's begin by defining **public** and **private**. When a member of a class is modified by **public**, then that member can be accessed by any other code. When a member of a class is specified as **private**, then that member can only be accessed by other members of its class. Now you can understand why **main()** has always been preceded by the **public** modifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package. (Packages are discussed in [Chapter 9](#).)

In the classes developed so far, all members of a class have used the default access mode. However, this is not what you will typically want to be the case. Usually, you will want to restrict access to the data members of a class—allowing access only through methods. Also, there will be times when you will want to define methods that are private to a class.

An access modifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here is an example:

```
public int i;  
private double j;  
  
private int myMethod(int a, char b) { //...
```

To understand the effects of public and private access, consider the following program:

```
/* This program demonstrates the difference between
   public and private.
*/
class Test {
    int a; // default access
    public int b; // public access
    private int c; // private access

    // methods to access c
    void setc(int i) { // set c's value
        c = i;
    }
    int getc() { // get c's value
        return c;
    }
}

class AccessTest {
    public static void main(String[] args) {
        Test ob = new Test();

        // These are OK, a and b may be accessed directly
        ob.a = 10;
        ob.b = 20;

        // This is not OK and will cause an error
        // ob.c = 100; // Error!

        // You must access c through its methods
        ob.setc(100); // OK
        System.out.println("a, b, and c: " + ob.a + " " +
                           ob.b + " " + ob.getc());
    }
}
```

As you can see, inside the **Test** class, **a** uses default access, which for this example is the same as specifying **public**. **b** is explicitly specified as **public**. Member **c** is given private access. This means that it cannot be accessed by code outside of its class. So, inside the **AccessTest** class, **c** cannot be used directly. It must be accessed through its public methods:

setc() and **getc()**). If you were to remove the comment symbol from the beginning of the following line,

```
// ob.c = 100; // Error!
```

then you would not be able to compile this program because of the access violation.

To see how access control can be applied to a more practical example, consider the following improved version of the **Stack** class shown at the end of [Chapter 6](#):

```
// This class defines an integer stack that can hold 10 values.
class Stack {
    /* Now, both stck and tos are private. This means
       that they cannot be accidentally or maliciously
       altered in a way that would be harmful to the stack.
    */
    private int[] stck = new int[10];
    private int tos;

    // Initialize top-of-stack
    Stack() {
        tos = -1;
    }

    // Push an item onto the stack
    void push(int item) {
        if(tos==9)
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }
}
```

```
// Pop an item from the stack
int pop() {
    if(tos < 0) {
        System.out.println("Stack underflow.");
        return 0;
    }
    else
        return stck[tos--];
}
}
```

As you can see, now both **stck**, which holds the stack, and **tos**, which is the index of the top of the stack, are specified as **private**. This means that they cannot be accessed or altered except through **push()** and **pop()**. Making **tos** private, for example, prevents other parts of your program from inadvertently setting it to a value that is beyond the end of the **stck** array.

The following program demonstrates the improved **Stack** class. Try removing the commented-out lines to prove to yourself that the **stck** and **tos** members are, indeed, inaccessible.

```
class TestStack {
    public static void main(String[] args) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();

        // push some numbers onto the stack
        for(int i=0; i<10; i++) mystack1.push(i);
        for(int i=10; i<20; i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<10; i++)
            System.out.println(mystack1.pop());

        System.out.println("Stack in mystack2:");

        for(int i=0; i<10; i++)
            System.out.println(mystack2.pop());

        // these statements are not legal
        // mystack1.tos = -2;
        // mystack2.stck[3] = 100;
    }
}
```

Although methods will usually provide access to the data defined by a class, this does not always have to be the case. It is perfectly proper to allow an instance variable to be public when there is good reason to do so. For example, most of the simple classes in this book were created with little concern about controlling access to instance variables for the sake of simplicity. However, in most real-world classes, you will need to allow operations on data only through methods. The next chapter will return to the topic of access control. As you will see, it is particularly important when inheritance is involved.

Understanding static

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member

must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be **static**. The most common example of a **static** member is **main()**. **main()** is declared as **static** because it must be called before any objects exist.

Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

Methods declared as **static** have several restrictions:

- They can only directly call other **static** methods of their class.
- They can only directly access **static** variables of their class.
- They cannot refer to **this** or **super** in any way. (The keyword **super** relates to inheritance and is described in the next chapter.)

If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block:

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
    static int a = 3;
    static int b;

    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }

    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }

    public static void main(String[] args) {
        meth(42);
    }
}
```

As soon as the **UseStatic** class is loaded, all of the **static** statements are run. First, **a** is set to **3**, then the **static** block executes, which prints a message and then initializes **b** to **a*4** or **12**. Then **main()** is called, which calls **meth()**, passing **42** to **x**. The three **println()** statements refer to the two **static** variables **a** and **b**, as well as to the parameter **x**.

Here is the output of the program:

```
Static block initialized.
x = 42
a = 3
b = 12
```

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. For example, if you wish to call a **static** method from outside its class, you can do so using the following general form:

classname.method()

Here, *classname* is the name of the class in which the **static** method is declared. As you can see, this format is similar to that used to call non-**static** methods through object-reference variables. A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.

Here is an example. Inside **main()**, the **static** method **callme()** and the **static** variable **b** are accessed through their class name **StaticDemo**.

```
class StaticDemo {  
    static int a = 42;  
    static int b = 99;  
  
    static void callme() {  
        System.out.println("a = " + a);  
    }  
}  
  
class StaticByName {  
    public static void main(String[] args) {  
        StaticDemo.callme();  
        System.out.println("b = " + StaticDemo.b);  
    }  
}
```

Here is the output of this program:

```
a = 42  
b = 99
```

Introducing **final**

A field can be declared as **final**. Doing so prevents its contents from being modified, making it, essentially, a constant. This means that you must initialize a **final** field when it is declared. You can do this in one of two ways: First, you can give it a value when it is declared. Second, you can assign it a value within a constructor. The first approach is probably the most common. Here is an example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Subsequent parts of your program can now use **FILE_OPEN**, etc., as if they were constants, without fear that a value has been changed. It is a common coding convention to choose all uppercase identifiers for **final** fields, as this example shows.

In addition to fields, both method parameters and local variables can be declared **final**. Declaring a parameter **final** prevents it from being changed within the method. Declaring a local variable **final** prevents it from being assigned a value more than once.

The keyword **final** can also be applied to methods, but its meaning is substantially different than when it is applied to variables. This additional usage of **final** is explained in the next chapter, when inheritance is described.

Arrays Revisited

Arrays were introduced earlier in this book, before classes had been discussed. Now that you know about classes, an important point can be made about arrays: they are implemented as objects. Because of this, there is a special array attribute that you will want to take advantage of. Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its **length** instance variable. All arrays have this variable, and it will always hold the size of the array. Here is a program that demonstrates this property:

```
// This program demonstrates the length array member.  
class Length {  
    public static void main(String[] args) {  
        int[] a1 = new int[10];  
        int[] a2 = {3, 5, 7, 1, 8, 99, 44, -10};  
        int[] a3 = {4, 3, 2, 1};  
  
        System.out.println("length of a1 is " + a1.length);  
        System.out.println("length of a2 is " + a2.length);  
        System.out.println("length of a3 is " + a3.length);  
    }  
}
```

This program displays the following output:

```
length of a1 is 10  
length of a2 is 8  
length of a3 is 4
```

As you can see, the size of each array is displayed. Keep in mind that the value of **length** has nothing to do with the number of elements that are actually in use. It only reflects the number of elements that the array is designed to hold.

You can put the **length** member to good use in many situations. For example, here is an improved version of the **Stack** class. As you might recall, the earlier versions of this class always created a ten-element stack. The following version lets you create stacks of any size. The value of **stck.length** is used to prevent the stack from overflowing.

```
// Improved Stack class that uses the length array member.
class Stack {
    private int[] stck;
    private int tos;

    // allocate and initialize stack
    Stack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // Push an item onto the stack
    void push(int item) {
        if(tos==stck.length-1) // use length member
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }

    // Pop an item from the stack
    int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}

class TestStack2 {
    public static void main(String[] args) {
        Stack mystack1 = new Stack(5);
        Stack mystack2 = new Stack(8);

        // push some numbers onto the stack
        for(int i=0; i<5; i++) mystack1.push(i);
        for(int i=0; i<8; i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());
    }
}
```

Notice that the program creates two stacks: one five elements deep and the other eight elements deep. As you can see, the fact that arrays maintain their own length information makes it easy to create stacks of any size.

```
for(int i=0; i<8; i++)
    System.out.println(mystack2.pop());
```

Introducing Nested and Inner Classes

It is possible to define a class within another class; such classes are known as *nested classes*. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.

There are two types of nested classes: *static* and *inner*. A static nested class is one that has the **static** modifier applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly.

The second type of nested class is the *inner* class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

The following program illustrates how to define and use an inner class. The class named **Outer** has one instance variable named **outer_x**, one instance method named **test()**, and defines one inner class called **Inner**.

```
// Demonstrate an inner class.
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // this is an inner class
    class Inner {
        void display() {
            System.out.println("display: outer_x = " + outer_x);
        }
    }
}

class InnerClassDemo {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Output from this application is shown here:

```
display: outer_x = 100
```

In the program, an inner class named **Inner** is defined within the scope of class **Outer**. Therefore, any code in class **Inner** can directly access the variable **outer_x**. An instance method named **display()** is defined inside **Inner**. This method displays **outer_x** on the standard output stream. The **main()** method of **InnerClassDemo** creates an instance of class **Outer** and invokes its **test()** method. That method creates an instance of class **Inner**, and the **display()** method is called.

It is important to realize that an instance of **Inner** can be created only in the context of class **Outer**. The Java compiler generates an error message otherwise. In general, an inner class instance is often created by code within its enclosing scope, as the example does.

As explained, an inner class has access to all of the members of its enclosing class, but the reverse is not true. Members of the inner class are known only within the scope of the inner class and may not be used by the outer class. For example:

```
// This program will not compile.
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // this is an inner class
    class Inner {
        int y = 10; // y is local to Inner

        void display() {
            System.out.println("display: outer_x = " + outer_x);
        }
    }

    void showy() {
        System.out.println(y); // error, y not known here!
    }
}

class InnerClassDemo {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Here, **y** is declared as an instance variable of **Inner**. Thus, it is not known outside of that class and it cannot be used by **showy()**.

Although we have been focusing on inner classes declared as members within an outer class scope, it is possible to define inner classes within any block scope. For example, you can define a nested class within the block

```
// Define an inner class within a for loop.
class Outer {
    int outer_x = 100;

    void test() {
        for(int i=0; i<10; i++) {
            class Inner {
                void display() {
                    System.out.println("display: outer_x = " + outer_x);
                }
            }
            Inner inner = new Inner();
            inner.display();
        }
    }
}

class InnerClassDemo {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

While nested classes are not applicable to all situations, they are particularly helpful when handling events. We will return to the topic of nested classes in [Chapter 25](#). There you will see how inner classes can be used to simplify the code needed to handle certain types of events. You will also learn about *anonymous inner classes*, which are inner classes that don't have a name.

One point of interest: Nested classes were not allowed by the original 1.0 specification for Java. They were added by Java 1.1.

Exploring the String Class

Although the **String** class will be examined in depth in Part II of this book, a short exploration of it is warranted now, because we will be using strings in some of the sample programs shown toward the end of Part I. **String** is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming.

The first thing to understand about strings is that every string you create is actually an object of type **String**. Even string constants are actually **String** objects. For example, in the statement

```
System.out.println("This is a String, too");
```

the string "This is a String, too" is a **String** object.

The second thing to understand about strings is that objects of type **String** are immutable; once a **String** object is created, its contents cannot be altered. While this may seem like a serious restriction, it is not, for two reasons:

- If you need to change a string, you can always create a new one that contains the modifications.
- Java defines peer classes of **String**, called **StringBuffer** and **StringBuilder**, that allow strings to be altered, so all of the normal string manipulations are still available in Java. (**StringBuffer** and **StringBuilder** are described in Part II of this book.)

Strings can be constructed in a variety of ways. The easiest is to use a statement like this:

```
String myString = "this is a test";
```

Once you have created a **String** object, you can use it anywhere that a string is allowed. For example, this statement displays **myString**:

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

Java defines one operator for **String** objects: **+**. It is used to concatenate two strings. For example, the statement

```
String myString = "I" + " like " + "Java.;"
```

results in **myString** containing "I like Java."

The following program demonstrates the preceding concepts:

```
// Demonstrating Strings.  
class StringDemo {  
    public static void main(String[] args) {  
        String strOb1 = "First String";  
        String strOb2 = "Second String";  
        String strOb3 = strOb1 + " and " + strOb2;  
  
        System.out.println(strOb1);  
  
        System.out.println(strOb2);  
        System.out.println(strOb3);  
    }  
}
```

The output produced by this program is shown here:

```
First String  
Second String  
First String and Second String
```

The **String** class contains several methods that you can use. Here are a few. You can test two strings for equality by using **equals()**. You can obtain the length of a string by calling the **length()** method. You can obtain the character at a specified index within a string by calling **charAt()**. The general forms of these three methods are shown here:

```
boolean equals(secondStr)
int length()
char charAt(index)
```

Here is a program that demonstrates these methods:

```
// Demonstrating some String methods.
class StringDemo2 {
    public static void main(String[] args) {
        String strOb1 = "First String";
        String strOb2 = "Second String";
        String strOb3 = strOb1;

        System.out.println("Length of strOb1: " +
                           strOb1.length());

        System.out.println("Char at index 3 in strOb1: " +
                           strOb1.charAt(3));

        if(strOb1.equals(strOb2))
            System.out.println("strOb1 == strOb2");
        else
            System.out.println("strOb1 != strOb2");

        if(strOb1.equals(strOb3))
            System.out.println("strOb1 == strOb3");
        else
            System.out.println("strOb1 != strOb3");
    }
}
```

This program generates the following output:

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

Of course, you can have arrays of strings, just like you can have arrays of any other type of object. For example:

```
// Demonstrate String arrays.
class StringDemo3 {
    public static void main(String[] args) {
        String[] str = { "one", "two", "three" };

        for(int i=0; i<str.length; i++)
            System.out.println("str[" + i + "]: " +
                               str[i]);
    }
}
```

Here is the output from this program:

```
str[0]: one  
str[1]: two  
str[2]: three
```

As you will see in the following section, string arrays play an important part in many Java programs.

Using Command-Line Arguments

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing *command-line arguments* to `main()`. A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy—they are stored as strings in a `String` array passed to the `args` parameter of `main()`. The first command-line argument is stored at `args[0]`, the second at `args[1]`, and so on. For example, the following program displays all of the command-line arguments that it is called with:

Try executing this program, as shown here:

```
java CommandLine this is a test 100 -1
```

When you do, you will see the following output:

```
args[0] : this
args[1] : is
args[2] : a
args[3] : test
args[4] : 100
args[5] : -1
```

REMEMBER All command-line arguments are passed as strings. You must convert numeric values to their internal forms manually, as explained in [Chapter 19](#).

Varargs: Variable-Length Arguments

Modern versions of Java include a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called *varargs*, and it is short for *variable-length arguments*. A method that takes a variable number of arguments is called a *variable-arity method*, or simply a *varargs method*.

Situations that require that a variable number of arguments be passed to a method are not unusual. For example, a method that opens an Internet connection might take a user name, password, filename, protocol, and so on, but supply defaults if some of this information is not provided. In this situation, it would be convenient to pass only the arguments to which the defaults did not apply. Another example is the `printf()` method that is part of Java's I/O library. As you will see in [Chapter 22](#), it takes a variable number of arguments, which it formats and then outputs.

In the early days of Java, variable-length arguments could be handled two ways, neither of which was particularly pleasing. First, if the maximum number of arguments was small and known, then you could create overloaded versions of the method, one for each way the method could be called. Although this works and is suitable for some cases, it applies to only a narrow class of situations.

In cases where the maximum number of potential arguments was larger, or unknowable, a second approach was used in which the arguments were put into an array, and then the array was passed to the method. This approach, which you may still find in older legacy code, is illustrated by the following program:

```
// Use an array to pass a variable number of
// arguments to a method. This is the old-style
// approach to variable-length arguments.
class PassArray {
    static void vaTest(int[] v) {
        System.out.print("Number of args: " + v.length +
                         " Contents: ");
        for(int x : v)
            System.out.print(x + " ");
        System.out.println();
    }

    public static void main(String[] args)
    {
        // Notice how an array must be created to
        // hold the arguments.
        int[] n1 = { 10 };
        int[] n2 = { 1, 2, 3 };
        int[] n3 = { };

        vaTest(n1); // 1 arg
        vaTest(n2); // 3 args
        vaTest(n3); // no args
    }
}
```

The output from the program is shown here:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

In the program, the method **vaTest()** is passed its arguments through the array **v**. This old-style approach to variable-length arguments does enable **vaTest()** to take an arbitrary number of arguments. However, it requires

that these arguments be manually packaged into an array prior to calling **vaTest()**. Not only is it tedious to construct an array each time **vaTest()** is called, it is potentially error-prone. The varargs feature offers a simpler, better option.

A variable-length argument is specified by three periods (...). For example, here is how **vaTest()** is written using a vararg:

```
static void vaTest(int ... v) {
```

This syntax tells the compiler that **vaTest()** can be called with zero or more arguments. As a result, **v** is implicitly declared as an array of type **int[]**. Thus, inside **vaTest()**, **v** is accessed using the normal array syntax. Here is the preceding program rewritten using a vararg:

```
// Demonstrate variable-length arguments.
class VarArgs {

    // vaTest() now uses a vararg.
    static void vaTest(int ... v) {
        System.out.print("Number of args: " + v.length +
                         " Contents: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    public static void main(String[] args)
    {
        // Notice how vaTest() can be called with a
        // variable number of arguments.
        vaTest(10);           // 1 arg
        vaTest(1, 2, 3);     // 3 args
        vaTest();             // no args
    }
}
```

The output from the program is the same as the original version.

There are two important things to notice about this program. First, as explained, inside **vaTest()**, **v** is operated on as an array. This is because **v** is an array. The ... syntax simply tells the compiler that a variable number of arguments will be used and that these arguments will be stored in the array referred to by **v**. Second, in **main()**, **vaTest()** is called with different numbers of arguments, including no arguments at all. The arguments are automatically put in an array and passed to **v**. In the case of no arguments, the length of the array is zero.

A method can have “normal” parameters along with a variable-length parameter. However, the variable-length parameter must be the last parameter declared by the method. For example, this method declaration is perfectly acceptable:

```
int doIt(int a, int b, double c, int ... vals) {
```

In this case, the first three arguments used in a call to **doIt()** are matched to the first three parameters. Then, any remaining arguments are assumed to belong to **vals**.

Remember, the varargs parameter must be last. For example, the following declaration is incorrect:

```
int doIt(int a, int b, double c, int ... vals, boolean
stopFlag) { // Error!
```

Here, there is an attempt to declare a regular parameter after the varargs parameter, which is illegal.

There is one more restriction to be aware of: there must be only one varargs parameter. For example, this declaration is also invalid:

```
int doIt(int a, int b, double c, int ... vals, double ...
morevals) { // Error!
```

The attempt to declare the second varargs parameter is illegal.

Here is a reworked version of the **vaTest()** method that takes a regular argument and a variable-length argument:

```
// Use varargs with standard arguments.
class VarArgs2 {

    // Here, msg is a normal parameter and v is a
    // varargs parameter.
    static void vaTest(String msg, int ... v) {
        System.out.print(msg + v.length +
                          " Contents: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    public static void main(String[] args)
    {
        vaTest("One vararg: ", 10);
        vaTest("Three varargs: ", 1, 2, 3);
        vaTest("No varargs: ");
    }
}
```

The output from this program is shown here:

```
One vararg: 1 Contents: 10
Three varargs: 3 Contents: 1 2 3
No varargs: 0 Contents:
```

Overloading Vararg Methods

You can overload a method that takes a variable-length argument. For example, the following program overloads `vaTest()` three times:

```
// Varargs and overloading.
class VarArgs3 {

    static void vaTest(int ... v) {
        System.out.print("vaTest(int ...): " +
                         "Number of args: " + v.length +
                         " Contents: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    static void vaTest(boolean ... v) {
        System.out.print("vaTest(boolean ...): " +
                         "Number of args: " + v.length +
                         " Contents: ");

        for(boolean x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    static void vaTest(String msg, int ... v) {
        System.out.print("vaTest(String, int ...): " +
                         msg + v.length +
                         " Contents: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    public static void main(String[] args)
    {
        vaTest(1, 2, 3);
        vaTest("Testing: ", 10, 20);
        vaTest(true, false, false);
    }
}
```

The output produced by this program is shown here:

```
vaTest(int ...): Number of args: 3 Contents: 1 2 3
vaTest(String, int ...): Testing: 2 Contents: 10 20
vaTest(boolean ...): Number of args: 3 Contents: true false false
```

This program illustrates both ways that a varargs method can be overloaded. First, the types of its vararg parameter can differ. This is the case for **vaTest(int ...)** and **vaTest(boolean ...)**. Remember, the ... causes the parameter to be treated as an array of the specified type. Therefore, just as you can overload methods by using different types of array parameters, you can overload vararg methods by using different types of varargs. In this case, Java uses the type difference to determine which overloaded method to call.

The second way to overload a varargs method is to add one or more normal parameters. This is what was done with **vaTest(String, int ...)**. In this case, Java uses both the number of arguments and the type of the arguments to determine which method to call.

NOTE A varargs method can also be overloaded by a non-varargs method. For example, **vaTest(int x)** is a valid overload of **vaTest()** in the foregoing program. This version is invoked only when one **int** argument is present. When two or more **int** arguments are passed, the varargs version **vaTest (int...v)** is used.

Varargs and Ambiguity

Somewhat unexpected errors can result when overloading a method that takes a variable-length argument. These errors involve ambiguity because it is possible to create an ambiguous call to an overloaded varargs method. For example, consider the following program:

```
// Varargs, overloading, and ambiguity.  
//  
// This program contains an error and will  
// not compile!  
class VarArgs4 {  
  
    static void vaTest(int ... v) {  
        System.out.print("vaTest(int ...): " +  
                         "Number of args: " + v.length +  
                         " Contents: ");  
  
        for(int x : v)  
            System.out.print(x + " ");  
  
        System.out.println();  
    }  
  
    static void vaTest(boolean ... v) {  
        System.out.print("vaTest(boolean ...): " +  
                         "Number of args: " + v.length +  
                         " Contents: ");  
  
        for(boolean x : v)  
            System.out.print(x + " ");  
  
        System.out.println();  
    }  
  
    public static void main(String[] args)  
    {  
        vaTest(1, 2, 3); // OK  
        vaTest(true, false, false); // OK  
  
        vaTest(); // Error: Ambiguous!  
    }  
}
```

In this program, the overloading of `vaTest()` is perfectly correct. However, this program will not compile because of the following call:

```
vaTest(); // Error: Ambiguous!
```

Because the vararg parameter can be empty, this call could be translated into a call to **vaTest(int ...)** or **vaTest(boolean ...)**. Both are equally valid. Thus, the call is inherently ambiguous.

Here is another example of ambiguity. The following overloaded versions of **vaTest()** are inherently ambiguous even though one takes a normal parameter:

```
static void vaTest(int ... v) { // ...
static void vaTest(int n, int ... v) { // ...
```

Although the parameter lists of **vaTest()** differ, there is no way for the compiler to resolve the following call:

```
vaTest(1)
```

Does this translate into a call to **vaTest(int ...)**, with one varargs argument, or into a call to **vaTest(int, int ...)** with no varargs arguments? There is no way for the compiler to answer this question. Thus, the situation is ambiguous.

Because of ambiguity errors like those just shown, sometimes you will need to forego overloading and simply use two different method names. Also, in some cases, ambiguity errors expose a conceptual flaw in your code, which you can remedy by more carefully crafting a solution.

Local Variable Type Inference with Reference Types

As you saw in [Chapter 3](#), beginning with JDK 10, Java supports local variable type inference. Recall that when using local variable type inference, the type of the variable is specified as **var** and the variable must be initialized. Earlier examples have shown type inference with primitive types, but it can also be used with reference types. In fact, type inference with reference types constitutes a primary use. Here is a simple example that declares a **String** variable called **myStr**:

```
var myStr = "This is a string";
```

Because a quoted string is used as an initializer, the type **String** is inferred.

As explained in [Chapter 3](#), one of the benefits of local variable type inference is its ability to streamline code, and it is with reference types where such streamlining is most apparent. The reason for this is that many class types in Java have rather long names. For example, in [Chapter 13](#), you will learn about the **FileInputStream** class, which is used to open a file for input operations. In the past, you would declare and initialize a **FileInputStream** using a traditional declaration like the one shown here:

```
FileInputStream fin = new FileInputStream("test.txt");
```

With the use of **var**, it can now be written like this:

```
var fin = new FileInputStream("test.txt");
```

Here, **fin** is inferred to be of type **FileInputStream** because that is the type of its initializer. There is no need to explicitly repeat the type name. As a result, this declaration of **fin** is substantially shorter than writing it the traditional way. Thus, the use of **var** streamlines the declaration. This benefit becomes even more apparent in more complex declarations, such as those involving generics. In general, the streamlining attribute of local variable type inference helps lessen the tedium of entering long type names into your program.

Of course, the streamlining aspect of local variable type inference must be used carefully to avoid reducing the readability of your program and, thus, obscuring its meaning. For example, consider a declaration such as the one shown here:

```
var x = o.getNext();
```

In this case, it may not be immediately clear to someone reading your code what the type of **x** is. In essence, local variable type inference is a feature that you should use wisely.

As you would expect, you can also use local variable type inference with user-defined classes, as the following program illustrates. It creates a class called **MyClass** and then uses local variable type inference to declare and initialize an object of that class.

```
// Local variable type inference with a user-defined class type.
class MyClass {
    private int i;

    MyClass(int k) { i = k; }

    int geti() { return i; }
    void seti(int k) { if(k >= 0) i = k; }
}

class RefVarDemo {
    public static void main(String[] args) {
        var mc = new MyClass(10); // Notice the use of var here.

        System.out.println("Value of i in mc is " + mc.geti());
        mc.seti(19);
        System.out.println("Value of i in mc is now " + mc.geti());
    }
}
```

The output of the program is shown here:

```
Value of i in mc is 10
Value of i in mc is now 19
```

In the program, pay special attention to this line:

```
var mc = new MyClass(10); // Notice the use of var here.
```

Here, the type of **mc** will be inferred as **MyClass** because that is the type of the initializer, which is a new **MyClass** object.

As explained earlier in this book, for the benefit of readers working in Java environments that do not support local variable type inference, it will not be used by most examples in the remainder of this edition of this book. This way, the majority of examples will compile and run for the largest number of readers.

CHAPTER

8

Inheritance

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a *superclass*. The class that does the inheriting is called a *subclass*. Therefore, a subclass is a specialized version of a superclass. It inherits all of the members defined by the superclass and adds its own, unique elements.

Inheritance Basics

In order to have your class inherit from a superclass, you simply incorporate the definition of the superclass into your class using the **extends** keyword. To see how, let's begin with a short example. The following program creates a superclass called **A** and a subclass called **B**. Notice how the keyword **extends** is used to create a subclass of **A**.

```
// A simple example of inheritance.

// Create a superclass.
class A {
    int i, j;

    void showij() {
        System.out.println("i and j: " + i + " " + j);
    }
}

// Create a subclass by extending class A.
class B extends A {
    int k;

    void showk() {
        System.out.println("k: " + k);
    }
}
```

```
void sum() {
    System.out.println("i+j+k: " + (i+j+k));
}
}

class SimpleInheritance {
    public static void main(String[] args) {
        A superOb = new A();
        B subOb = new B();

        // The superclass may be used by itself.
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Contents of superOb: ");
        superOb.showij();
        System.out.println();

        /* The subclass has access to all public members of
           its superclass. */
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println("Contents of subOb: ");
        subOb.showij();
        subOb.showk();
        System.out.println();

        System.out.println("Sum of i, j and k in subOb:");
        subOb.sum();
    }
}
```

The output from this program is shown here:

```
Contents of superOb:  
i and j: 10 20
```

```
Contents of subOb:  
i and j: 7 8  
k: 9
```

```
Sum of i, j and k in subOb:  
i+j+k: 24
```

As you can see, the subclass **B** includes all of the members of its superclass, **A**. This is why **subOb** can access **i** and **j** and call **showij()**. Also, inside **sum()**, **i** and **j** can be referred to directly, as if they were part of **B**.

Even though **A** is a superclass for **B**, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a superclass for another subclass.

The general form of a **class** declaration that inherits a superclass is shown here:

```
class subclass-name extends superclass-name {  
    // body of class  
}
```

You can only specify one superclass for any subclass that you create. Java does not support the inheritance of multiple superclasses into a single subclass. You can, as stated, create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**. For example, consider the following simple class hierarchy:

```
/* In a class hierarchy, private members remain
private to their class.

This program contains an error and will not
compile.
*/
// Create a superclass.
class A {
    int i; // default access
    private int j; // private to A

    void setij(int x, int y) {
        i = x;
        j = y;
    }
}

// A's j is not accessible here.
class B extends A {
    int total;

    void sum() {
        total = i + j; // ERROR, j is not accessible here
    }
}

class Access {
    public static void main(String[] args) {
        B subOb = new B();

        subOb.setij(10, 12);

        subOb.sum();
        System.out.println("Total is " + subOb.total);
    }
}
```

This program will not compile because the use of **j** inside the **sum()** method of **B** causes an access violation. Since **j** is declared as **private**, it is

only accessible by other members of its own class. Subclasses have no access to it.

REMEMBER A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

A More Practical Example

Let's look at a more practical example that will help illustrate the power of inheritance. Here, the final version of the **Box** class developed in the preceding chapter will be extended to include a fourth component called **weight**. Thus, the new class will contain a box's width, height, depth, and weight.

```
// This program uses inheritance to extend Box.
class Box {
    double width;
    double height;
    double depth;

    // construct clone of an object
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

// Here, Box is extended to include weight.
class BoxWeight extends Box {
```

```
double weight; // weight of box

// constructor for BoxWeight
BoxWeight(double w, double h, double d, double m) {
    width = w;
    height = h;
    depth = d;
    weight = m;
}
}

class DemoBoxWeight {
    public static void main(String[] args) {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
        double vol;

        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        System.out.println("Weight of mybox1 is " + mybox1.weight);
        System.out.println();

        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        System.out.println("Weight of mybox2 is " + mybox2.weight);
    }
}
```

The output from this program is shown here:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
```

```
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

BoxWeight inherits all of the characteristics of **Box** and adds to them the **weight** component. It is not necessary for **BoxWeight** to re-create all of the features found in **Box**. It can simply extend **Box** to meet its own purposes.

A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses. Each subclass can precisely tailor its own classification. For example, the following class inherits **Box** and adds a color attribute:

```
// Here, Box is extended to include color.
class ColorBox extends Box {
    int color; // color of box

    ColorBox(double w, double h, double d, int c) {
        width = w;
        height = h;
        depth = d;
        color = c;
    }
}
```

Remember, once you have created a superclass that defines the general aspects of an object, that superclass can be inherited to form specialized classes. Each subclass simply adds its own unique attributes. This is the essence of inheritance.

A Superclass Variable Can Reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. You will find this aspect of inheritance quite useful in a variety of situations. For example, consider the following:

```

class RefDemo {
    public static void main(String[] args) {
        BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
        Box plainbox = new Box();
        double vol;

        vol = weightbox.volume();
        System.out.println("Volume of weightbox is " + vol);
        System.out.println("Weight of weightbox is " +
                           weightbox.weight);
        System.out.println();

        // assign BoxWeight reference to Box reference
        plainbox = weightbox;

        vol = plainbox.volume(); // OK, volume() defined in Box
        System.out.println("Volume of plainbox is " + vol);

        /* The following statement is invalid because plainbox
           does not define a weight member. */
        // System.out.println("Weight of plainbox is " + plainbox.weight);
    }
}

```

Here, **weightbox** is a reference to **BoxWeight** objects, and **plainbox** is a reference to **Box** objects. Since **BoxWeight** is a subclass of **Box**, it is permissible to assign **plainbox** a reference to the **weightbox** object.

It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed. That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass. This is why **plainbox** can't access **weight** even when it refers to a **BoxWeight** object. If you think about it, this makes sense, because the superclass has no knowledge of what a subclass adds to it. This is why the last line of code in the preceding fragment is commented out. It is not possible for a **Box** reference to access the **weight** field, because **Box** does not define one.

Although the preceding may seem a bit esoteric, it has some important practical applications—two of which are discussed later in this chapter.

Using super

In the preceding examples, classes derived from **Box** were not implemented as efficiently or as robustly as they could have been. For example, the constructor for **BoxWeight** explicitly initializes the **width**, **height**, and **depth** fields of **Box**. Not only does this duplicate code found in its superclass, which is inefficient, but it implies that a subclass must be granted access to these members. However, there will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own. Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.

super has two general forms. The first calls the superclass's constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass. Each use is examined here.

Using super to Call Superclass Constructors

A subclass can call a constructor defined by its superclass by use of the following form of **super**:

```
super(arg-list);
```

Here, *arg-list* specifies any arguments needed by the constructor in the superclass. **super()** must always be the first statement executed inside a subclass's constructor.

To see how **super()** is used, consider this improved version of the **BoxWeight** class:

```
// BoxWeight now uses super to initialize its Box attributes.  
class BoxWeight extends Box {  
    double weight; // weight of box  
  
    // initialize width, height, and depth using super()  
    BoxWeight(double w, double h, double d, double m) {  
        super(w, h, d); // call superclass constructor  
        weight = m;  
    }  
}
```

Here, **BoxWeight()** calls **super()** with the arguments **w**, **h**, and **d**. This causes the **Box** constructor to be called, which initializes **width**, **height**, and **depth** using these values. **BoxWeight** no longer initializes these values itself. It only needs to initialize the value unique to it: **weight**. This leaves **Box** free to make these values **private** if desired.

In the preceding example, **super()** was called with three arguments. Since constructors can be overloaded, **super()** can be called using any form defined by the superclass. The constructor executed will be the one that matches the arguments. For example, here is a complete implementation of **BoxWeight** that provides constructors for the various ways that a box can be constructed. In each case, **super()** is called using the appropriate arguments. Notice that **width**, **height**, and **depth** have been made private within **Box**.

```
// A complete implementation of BoxWeight.
class Box {
    private double width;
    private double height;
    private double depth;

    // construct clone of an object
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
    double weight; // weight of box
```

```
super(w, h, d); // call superclass constructor
weight = m;
}

// default constructor
BoxWeight() {
    super();
    weight = -1;
}

// constructor used when cube is created
BoxWeight(double len, double m) {
    super(len);
    weight = m;
}
}

class DemoSuper {
    public static void main(String[] args) {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
        BoxWeight mybox3 = new BoxWeight(); // default
        BoxWeight mycube = new BoxWeight(3, 2);
        BoxWeight myclone = new BoxWeight(mybox1);
        double vol;

        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        System.out.println("Weight of mybox1 is " + mybox1.weight);
        System.out.println();

        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        System.out.println("Weight of mybox2 is " + mybox2.weight);
        System.out.println();

        vol = mybox3.volume();
        System.out.println("Volume of mybox3 is " + vol);
        System.out.println("Weight of mybox3 is " + mybox3.weight);
        System.out.println();

        vol = myclone.volume();
    }
}
```

```

This program generates the following output:
System.out.println("Volume of myclone is " + vol);
System.out.println("Weight of myclone is " + myclone.weight);
Weight of mybox1 is 34.3

Volume of mybox2 is 24.0      mycube is " + vol);
Weight of mybox2 is 0.076    mycube is " + mycube.weight);

} Volume of mybox3 is -1.0
} Weight of mybox3 is -1.0

Volume of myclone is 3000.0
Weight of myclone is 34.3

Volume of mycube is 27.0
Weight of mycube is 2.0

```

Pay special attention to this constructor in **BoxWeight**:

```

// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
    super(ob);
    weight = ob.weight;
}

```

Notice that **super()** is passed an object of type **BoxWeight**—not of type **Box**. This still invokes the constructor **Box(Box ob)**. As mentioned earlier, a superclass variable can be used to reference any object derived from that class. Thus, we are able to pass a **BoxWeight** object to the **Box** constructor. Of course, **Box** only has knowledge of its own members.

Let's review the key concepts behind **super()**. When a subclass calls **super()**, it is calling the constructor of its immediate superclass. Thus, **super()** always refers to the superclass immediately above the calling class. This is true even in a multileveled hierarchy. Also, **super()** must always be the first statement executed inside a subclass constructor.

A Second Use for **super**

The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

`super.member`

Here, *member* can be either a method or an instance variable.

This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:

```
// Using super to overcome name hiding.  
class A {  
    int i;  
}  
  
// Create a subclass by extending class A.  
  
class B extends A {  
    int i; // this i hides the i in A  
  
    B(int a, int b) {  
        super.i = a; // i in A  
        i = b; // i in B  
    }  
  
    void show() {  
        System.out.println("i in superclass: " + super.i);  
        System.out.println("i in subclass: " + i);  
    }  
}  
  
class UseSuper {  
    public static void main(String[] args) {  
        B subOb = new B(1, 2);  
  
        subOb.show();  
    }  
}
```

This program displays the following:

```
i in superclass: 1  
i in subclass: 2
```

Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass. As you will see, **super** can also be used to call methods that are hidden by a subclass.

Creating a Multilevel Hierarchy

Up to this point, we have been using simple class hierarchies that consist of only a superclass and a subclass. However, you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another. For example, given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is a subclass of **A**. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, **C** inherits all aspects of **B** and **A**. To see how a multilevel hierarchy can be useful, consider the following program. In it, the subclass **BoxWeight** is used as a superclass to create the subclass called **Shipment**. **Shipment** inherits all of the traits of **BoxWeight** and **Box**, and it adds a field called **cost**, which holds the cost of shipping such a parcel.

```
// Extend BoxWeight to include shipping costs.  
  
// Start with Box.  
class Box {  
    private double width;  
    private double height;  
    private double depth;
```

```
// construct clone of an object
Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
}

// constructor used when all dimensions specified
Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
}

// constructor used when no dimensions specified
Box() {
    width = -1; // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1; // box
}

// constructor used when cube is created
Box(double len) {
    width = height = depth = len;
}

// compute and return volume
double volume() {
    return width * height * depth;
}

// Add weight.
class BoxWeight extends Box {
    double weight; // weight of box

    // construct clone of an object
    BoxWeight(BoxWeight ob) { // pass object to constructor
        super(ob);
        weight = ob.weight;
    }
}
```

```
// constructor used when cube is created
BoxWeight(double len, double m) {
    super(len);
    weight = m;
}
}

// Add shipping costs.
class Shipment extends BoxWeight {
    double cost;

    // construct clone of an object
    Shipment(Shipment ob) { // pass object to constructor
        super(ob);
        cost = ob.cost;
    }

    // constructor when all parameters are specified
    Shipment(double w, double h, double d,
             double m, double c) {
        super(w, h, d, m); // call superclass constructor
        cost = c;
    }

    // default constructor
    Shipment() {
        super();
        cost = -1;
    }

    // constructor used when cube is created
    Shipment(double len, double m, double c) {
        super(len, m);
        cost = c;
    }
}

class DemoShipment {
    public static void main(String[] args) {
        Shipment shipment1 =
            new Shipment(10, 20, 15, 10, 3.41);
        Shipment shipment2 =

```

```

        vol = shipment2.volume();
        System.out.println("Volume of shipment2 is " + vol);
        System.out.println("Weight of shipment2 is "
                           + shipment2.weight);
        System.out.println("Shipping cost: $" + shipment2.cost);
    }
}

System.out.println("Weight of shipment1 is "
                   + shipment1.weight);
System.out.println("Shipping cost: $" + shipment1.cost);
System.out.println("Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41

Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28

```

Because of inheritance, **Shipment** can make use of the previously defined classes of **Box** and **BoxWeight**, adding only the extra information it needs for its own, specific application. This is part of the value of inheritance; it allows the reuse of code.

This example illustrates one other important point: **super()** always refers to the constructor in the closest superclass. The **super()** in **Shipment** calls the constructor in **BoxWeight**. The **super()** in **BoxWeight** calls the constructor in **Box**. In a class hierarchy, if a superclass constructor requires arguments, then all subclasses must pass those arguments “up the line.” This is true whether or not a subclass needs arguments of its own.

NOTE In the preceding program, the entire class hierarchy, including **Box**, **BoxWeight**, and **Shipment**, is shown all in one file. This is for your convenience only. In Java, all three classes could have been placed into their own files and compiled separately. In fact, using separate files is the norm, not the exception, in creating class hierarchies.

When Constructors Are Executed

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy executed? For example, given a subclass called **B** and a superclass called **A**, is **A**’s constructor executed before **B**’s, or vice versa? The answer is that in a class hierarchy, constructors complete

their execution in order of derivation, from superclass to subclass. Further, since **super()** must be the first statement executed in a subclass's constructor, this order is the same whether or not **super()** is used. If **super()** is not used, then the default or parameterless constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```
// Demonstrate when constructors are executed.

// Create a super class.
class A {
    A() {
        System.out.println("Inside A's constructor.");
    }
}

// Create a subclass by extending class A.
class B extends A {
    B() {
        System.out.println("Inside B's constructor.");
    }
}

// Create another subclass by extending B.
class C extends B {
    C() {
        System.out.println("Inside C's constructor.");
    }
}

class CallingCons {
    public static void main(String[] args) {
        C c = new C();
    }
}
```

The output from this program is shown here:

```
Inside A's constructor  
Inside B's constructor  
Inside C's constructor
```

As you can see, the constructors are executed in order of derivation.

If you think about it, it makes sense that constructors complete their execution in order of derivation. Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must complete its execution first.

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called through its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden. Consider the following:

```
// Method overriding.  
class A {  
    int i, j;  
    A(int a, int b) {  
        i = a;  
        j = b;  
    }  
  
    // display i and j  
    void show() {  
        System.out.println("i and j: " + i + " " + j);  
    }  
}
```

```
class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    // display k - this overrides show() in A
    void show() {
        System.out.println("k: " + k);
    }
}

class Override {
    public static void main(String[] args) {
        B subOb = new B(1, 2, 3);

        subOb.show(); // this calls show() in B
    }
}
```

The output produced by this program is shown here:

```
k: 3
```

When **show()** is invoked on an object of type **B**, the version of **show()** defined within **B** is used. That is, the version of **show()** inside **B** overrides the version declared in **A**.

If you wish to access the superclass version of an overridden method, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show()** is invoked within the subclass's version. This allows all instance variables to be displayed.

```
class B extends A {  
    int k;  
  
    B(int a, int b, int c) {  
        super(a, b);  
        k = c;  
    }  
  
    void show() {  
        super.show(); // this calls A's show()  
        System.out.println("k: " + k);  
    }  
}
```

If you substitute this version of **A** into the previous program, you will see the following output:

```
i and j: 1 2  
k: 3
```

Here, **super.show()** calls the superclass version of **show()**.

Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded. For example, consider this modified version of the preceding example:

```
// Methods with differing type signatures are overloaded - not
// overridden.
class A {
    int i, j;

    A(int a, int b) {
        i = a;
        j = b;
    }

    // display i and j
    void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}

// Create a subclass by extending class A.
class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    // overload show()
    void show(String msg) {
        System.out.println(msg + k);
    }
}

class Override {
    public static void main(String[] args) {
        B subOb = new B(1, 2, 3);

        subOb.show("This is k: "); // this calls show() in B
        subOb.show(); // this calls show() in A
    }
}
```

The output produced by this program is shown here:

```
This is k: 3  
i and j: 1 2
```

The version of **show()** in **B** takes a string parameter. This makes its type signature different from the one in **A**, which takes no parameters. Therefore, no overriding (or name hiding) takes place. Instead, the version of **show()** in **B** simply overloads the version of **show()** in **A**.

Dynamic Method Dispatch

While the examples in the preceding section demonstrate the mechanics of method overriding, they do not show its power. Indeed, if there were nothing more to method overriding than a name space convention, then it would be, at best, an interesting curiosity, but of little real value. However, this is not the case. Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch*. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Let's begin by restating an important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Here is an example that illustrates dynamic method dispatch:

```
// Dynamic Method Dispatch
class A {
    void callme() {
        System.out.println("Inside A's callme method");
    }
}

class B extends A {
    // override callme()
    void callme() {
        System.out.println("Inside B's callme method");
    }
}

class C extends A {
    // override callme()
    void callme() {
        System.out.println("Inside C's callme method");
    }
}

class Dispatch {
    public static void main(String[] args) {
        A a = new A(); // object of type A
        B b = new B(); // object of type B
        C c = new C(); // object of type C

        A r; // obtain a reference of type A

        r = a; // r refers to an A object
        r.callme(); // calls A's version of callme

        r = b; // r refers to a B object
        r.callme(); // calls B's version of callme

        r = c; // r refers to a C object
        r.callme(); // calls C's version of callme
    }
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

This program creates one superclass called **A** and two subclasses of it, called **B** and **C**. Subclasses **B** and **C** override **callme()** declared in **A**. Inside the **main()** method, objects of type **A**, **B**, and **C** are declared. Also, a reference of type **A**, called **r**, is declared. The program then in turn assigns a reference to each type of object to **r** and uses that reference to invoke **callme()**. As the output shows, the version of **callme()** executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, **r**, you would see three calls to **A**'s **callme()** method.

NOTE Readers familiar with C++ or C# will recognize that overridden methods in Java are similar to virtual functions in those languages.

Why Overridden Methods?

As stated earlier, overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.

Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy that moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface. Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.

Dynamic, run-time polymorphism is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness. The ability of existing code libraries to call methods on

instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

Applying Method Overriding

Let's look at a more practical example that uses method overriding. The following program creates a superclass called **Figure** that stores the dimensions of a two-dimensional object. It also defines a method called **area()** that computes the area of an object. The program derives two subclasses from **Figure**. The first is **Rectangle** and the second is **Triangle**. Each of these subclasses overrides **area()** so that it returns the area of a rectangle and a triangle, respectively.

```
// Using run-time polymorphism.
class Figure {
    double dim1;
    double dim2;

    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    double area() {
        System.out.println("Area for Figure is undefined.");
        return 0;
    }
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // override area for rectangle
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}

class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // override area for right triangle
    double area() {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}

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

```
Triangle t = new Triangle(10, 8);
Figure figref;

figref = r;
System.out.println("Area is " + figref.area());

figref = t;
System.out.println("Area is " + figref.area());

figref = f;
System.out.println("Area is " + figref.area());
}
}
```

The output from the program is shown here:

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

Through the dual mechanisms of inheritance and run-time polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects. In this case, if an object is derived from **Figure**, then its area can be obtained by calling **area()**. The interface to this operation is the same no matter what type of figure is being used.

Using Abstract Classes

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable

to create a meaningful implementation for a method. This is the case with the class **Figure** used in the preceding example. The definition of **area()** is simply a placeholder. It will not compute and display the area of any type of object.

As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have methods that must be overridden by the subclass in order for the subclass to have any meaning. Consider the class **Triangle**. It has no meaning if **area()** is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the *abstract method*.

You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass. To declare an abstract method, use this general form:

```
abstract type name(parameter-list);
```

As you can see, no method body is present.

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass or be declared **abstract** itself.

Here is a simple example of a class with an abstract method, followed by a class which implements that method:

```
// A Simple demonstration of abstract.  
abstract class A {  
    abstract void callme();  
  
    // concrete methods are still allowed in abstract classes  
    void callmetoo() {  
        System.out.println("This is a concrete method.");  
    }  
}  
  
class B extends A {  
    void callme() {  
        System.out.println("B's implementation of callme.");  
    }  
}  
  
class AbstractDemo {  
    public static void main(String[] args) {  
        B b = new B();  
  
        b.callme();  
        b.callmetoo();  
    }  
}
```

Notice that no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class **A** implements a concrete method called **callmetoo()**. This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object. You will see this feature put to use in the next example.

Using an abstract class, you can improve the **Figure** class shown earlier. Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the program declares **area()** as

abstract inside **Figure**. This, of course, means that all classes derived from **Figure** must override **area()**.

```
// Using abstract methods and classes.
abstract class Figure {
    double dim1;
    double dim2;

    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    // area is now an abstract method
    abstract double area();
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // override area for rectangle
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}

class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // override area for right triangle
    double area() {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}

class AbstractAreas {
    public static void main(String[] args) {
        // Figure f = new Figure(10, 10); // illegal now
        Rectangle r = new Rectangle(10, 10).
```

```
    figref = t;
    System.out.println("Area is " + figref.area());ated
}
}
figref = r;
System.out.println("Area is " + figref.area());
```

As the comment inside **main()** indicates, it is no longer possible to declare objects of type **Figure**, since it is now abstract. And, all subclasses of **Figure** must override **area()**. To prove this to yourself, try creating a subclass that does not override **area()**. You will receive a compile-time error.

Although it is not possible to create an object of type **Figure**, you can create a reference variable of type **Figure**. The variable **figref** is declared as a reference to **Figure**, which means that it can be used to refer to an object of any class derived from **Figure**. As explained, it is through superclass reference variables that overridden methods are resolved at run time.

Using final with Inheritance

The keyword **final** has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of **final** apply to inheritance. Both are examined here.

Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden. The following fragment illustrates **final**:

```

class A {
    final void meth() {
        System.out.println("This is a final method.");
    }
}

class B extends A {
    void meth() { // ERROR! Can't override.
        System.out.println("Illegal!");
    }
}

```

Because **meth()** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.

Methods declared as **final** can sometimes provide a performance enhancement: The compiler is free to *inline* calls to them because it “knows” they will not be overridden by a subclass. When a small **final** method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is an option only with **final** methods. Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*. However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called *early binding*.

Using **final** to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a **final** class:

```
final class A {  
    //...  
}  
  
// The following class is illegal.  
class B extends A { // ERROR! Can't subclass A  
    //...  
}
```

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

NOTE Beginning with JDK 17, the ability to *seal* a class was added to Java. Sealing offers fine-grained control over inheritance. Sealing is described in [Chapter 17](#).

Local Variable Type Inference and Inheritance

As explained in [Chapter 3](#), JDK 10 added local variable type inference to the Java language, which is supported by the context-sensitive keyword **var**. It is important to have a clear understanding of how type inference works within an inheritance hierarchy. Recall that a superclass reference can refer to a derived class object, and this feature is part of Java's support for polymorphism. However, it is critical to remember that when using local variable type inference, the inferred type of a variable is based on the declared type of its initializer. Therefore, if the initializer is of the superclass type, that will be the inferred type of the variable. It does not matter if the actual object being referred to by the initializer is an instance of a derived class. For example, consider this program:

```
// When working with inheritance, the inferred type is the declared
// type of the initializer, which may not be the most derived type of
// the object being referred to by the initializer.

class MyClass {
    // ...
}

class FirstDerivedClass extends MyClass {
    int x;
    // ...
}
```

```
class SecondDerivedClass extends FirstDerivedClass {
    int y;
    // ...
}

class TypeInferenceAndInheritance {

    // Return some type of MyClass object.
    static MyClass getObj(int which) {
        switch(which) {
            case 0: return new MyClass();
            case 1: return new FirstDerivedClass();
            default: return new SecondDerivedClass();
        }
    }

    public static void main(String[] args) {

        // Even though getObj() returns different types of
        // objects within the MyClass inheritance hierarchy,
        // its declared return type is MyClass. As a result,
        // in all three cases shown here, the type of the
        // variables is inferred to be MyClass, even though
        // different derived types of objects are obtained.

        // Here, getObj() returns a MyClass object.
        var mc = getObj(0);

        // In this case, a FirstDerivedClass object is returned.
        var mc2 = getObj(1);

        // Here, a SecondDerivedClass object is returned.
        var mc3 = getObj(2);

        // Because the types of both mc2 and mc3 are inferred
        // as MyClass (because the return type of getObj() is
        // MyClass), neither mc2 nor mc3 can access the fields
        // declared by FirstDerivedClass or SecondDerivedClass.
        // mc2.x = 10; // Wrong! MyClass does not have an x field.
        // mc3.y = 10; // Wrong! MyClass does not have a y field.
    }
}
```

In the program, a hierarchy is created that consists of three classes, at the top of which is **MyClass**. **FirstDerivedClass** is a subclass of **MyClass**, and **SecondDerivedClass** is a subclass of **FirstDerivedClass**. The program then uses type inference to create three variables, called **mc**, **mc2**, and **mc3**, by calling **getObj()**. The **getObj()** method has a return type of **MyClass** (the superclass) but returns objects of type **MyClass**, **FirstDerivedClass**, or **SecondDerivedClass**, depending on the argument that it is passed. As the output shows, the inferred type is determined by the return type of **getObj()**, not by the actual type of the object obtained. Thus, all three variables will be of type **MyClass**.

The Object Class

There is one special class, **Object**, defined by Java. All other classes are subclasses of **Object**. That is, **Object** is a superclass of all other classes. This means that a reference variable of type **Object** can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type **Object** can also refer to any array.

Object defines the following methods, which means that they are available in every object.

Method	Purpose
Object clone()	Creates a new object that is the same as the object being cloned.
boolean equals(Object <i>object</i>)	Determines whether one object is equal to another.
void finalize()	Called before an unused object is recycled. (Deprecated for removal by JDK 18.)
Class<?> getClass()	Obtains the class of an object at run time.
int hashCode()	Returns the hash code associated with the invoking object.
void notify()	Resumes execution of a thread waiting on the invoking object.
void notifyAll()	Resumes execution of all threads waiting on the invoking object.
String toString()	Returns a string that describes the object.
void wait() void wait(long <i>milliseconds</i>) void wait(long <i>milliseconds</i> , int <i>nanoseconds</i>)	Waits on another thread of execution.

The methods **getClass()**, **notify()**, **notifyAll()**, and **wait()** are declared as **final**. You may override the others. These methods are described elsewhere in this book. However, notice two methods now: **equals()** and **toString()**. The **equals()** method compares two objects. It returns **true** if the objects are equal, and **false** otherwise. The precise definition of equality can vary, depending on the type of objects being compared. The **toString()** method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using **println()**. Many classes override this method. Doing so allows them to tailor a description specifically for the types of objects that they create.

One last point: Notice the unusual syntax in the return type for **getClass()**. This relates to Java's *generics* feature, which is described in [Chapter 14](#).

CHAPTER

9

Packages and Interfaces

This chapter examines two of Java’s most innovative features: packages and interfaces. *Packages* are containers for classes. They are used to keep the class name space compartmentalized. For example, a package allows you to create a class named **List**, which you can store in your own package without concern that it will collide with some other class named **List** stored elsewhere. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions. As you will see in [Chapter 16](#), packages also play an important role with modules.

In previous chapters, you have seen how methods define the interface to the data in a class. Through the use of the **interface** keyword, Java allows you to fully abstract an interface from its implementation. Using **interface**, you can specify a set of methods that can be implemented by one or more classes. In its traditional form, the **interface** itself does not actually define any implementation. Although they are similar to abstract classes, **interfaces** have an additional capability: A class can implement more than one interface. By contrast, a class can only inherit a single superclass (abstract or otherwise).

Packages

In the preceding chapters, the name of each sample class was taken from the same name space. This means that a unique name had to be used for each class to avoid name collisions. After a while, without some way to manage the name space, you could run out of convenient, descriptive names for individual classes. You also need some way to be assured that the

name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers. (Imagine a small group of programmers fighting over who gets to use the name “Foobar” as a class name. Or, imagine the entire Internet community arguing over who first named a class “Espresso.”) Thankfully, Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are exposed only to other members of the same package. This allows your classes to have intimate knowledge of each other but not expose that knowledge to the rest of the world.

Defining a Package

To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name. (This is why you haven’t had to worry about packages before now.) While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

This is the general form of the **package** statement:

```
package pkg;
```

Here, *pkg* is the name of the package. For example, the following statement creates a package called **mypackage**:

```
package mypackage;
```

Typically, Java uses file system directories to store packages, and that is the approach assumed by the examples in this book. For example, the **.class** files for any classes you declare to be part of **mypackage** must be stored in a directory called **mypackage**. Remember that case is significant, and the directory name must match the package name exactly.

More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

```
package pkg1[pkg2[pkg3]];
```

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

```
package a.b.c;
```

needs to be stored in **a\b\c** in a Windows environment. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH

As just explained, packages are typically mirrored by directories. This raises an important question: How does the Java run-time system know where to look for packages that you create? As it relates to the examples in this chapter, the answer has three parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable. Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes. It is useful to point out that, beginning with JDK 9, a package can be part of a module, and thus found on the module path. However, a discussion of modules and module paths is deferred until [Chapter 16](#). For now, we will use only class paths.

For example, consider the following package specification:

```
package mypack;
```

In order for a program to find **mypad**, the program can be executed from a directory immediately above **mypad**, or the **CLASSPATH** must be set to include the path to **mypad**, or the **-classpath** option must specify the path to **mypad** when the program is run via **java**.

When the last two options are used, the class path *must not* include **mypad** itself. It must simply specify the *path to mypad*. For example, in a Windows environment, if the path to **mypad** is

C:\MyPrograms\Java\mypad

then the class path to **mypad** is

C:\MyPrograms\Java

The easiest way to try the examples shown in this book is to simply create the package directories below your current development directory, put the **.class** files into the appropriate directories, and then execute the programs from the development directory. This is the approach used in the following example.

A Short Package Example

Keeping the preceding discussion in mind, you can try this simple package:

```
// A simple package
package mypack;

class Balance {
    String name;
    double bal;

    Balance(String n, double b) {
        name = n;
        bal = b;
    }

    void show() {
        if(bal<0)
            System.out.print("-> ");
        System.out.println(name + ": $" + bal);
    }
}

class AccountBalance {
    public static void main(String[] args) {
        Balance[] current = new Balance[3];

        current[0] = new Balance("K. J. Fielding", 123.23);
        current[1] = new Balance("Will Tell", 157.02);
        current[2] = new Balance("Tom Jackson", -12.33);

        for(int i=0; i<3; i++) current[i].show();
    }
}
```

Call this file **AccountBalance.java** and put it in a directory called **mypack**.

Next, compile the file. Make sure that the resulting **.class** file is also in the **mypack** directory. Then, try executing the **AccountBalance** class, using the following command line:

```
java mypack.AccountBalance
```

Remember, you will need to be in the directory above **mypack** when you execute this command. (Alternatively, you can use one of the other two

options described in the preceding section to specify the path **mypack**.)

As explained, **AccountBalance** is now part of the package **mypack**. This means that it cannot be executed by itself. That is, you cannot use this command line:

```
java AccountBalance
```

AccountBalance must be qualified with its package name.

Packages and Member Access

In the preceding chapters, you learned about various aspects of Java's access control mechanism and its access modifiers. For example, you already know that access to a **private** member of a class is granted only to other members of that class. Packages add another dimension to access control. As you will see, Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. As it relates to the interplay between classes and packages, Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three access modifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories. [Table 9-1](#) sums up the interactions.

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Table 9-1 Class Member Access

While Java’s access control mechanism may seem complicated, we can simplify it as follows. Anything declared **public** can be accessed from different classes and different packages. Anything declared **private** cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.

Table 9-1 applies only to members of classes. A non-nested class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible outside its package. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.

NOTE The modules feature can also affect accessibility. Modules are described in [Chapter 16](#).

An Access Example

The following example shows all combinations of the access control modifiers. This example has two packages and five classes. Remember that the classes for the two different packages need to be stored in directories named after their respective packages—in this case, **p1** and **p2**.

The source for the first package defines three classes: **Protection**, **Derived**, and **SamePackage**. The first class defines four **int** variables in each of the legal protection modes. The variable **n** is declared with the

default protection, **n_pri** is **private**, **n_pro** is **protected**, and **n_pub** is **public**.

Each subsequent class in this example will try to access the variables in an instance of this class. The lines that will not compile due to access restrictions are commented out. Before each of these lines is a comment listing the places from which this level of protection would allow access.

The second class, **Derived**, is a subclass of **Protection** in the same package, **p1**. This grants **Derived** access to every variable in **Protection** except for **n_pri**, the **private** one. The third class, **SamePackage**, is not a subclass of **Protection**, but it is in the same package and also has access to all but **n_pri**.

This is file **Protection.java**:

```
package p1;

public class Protection {
    int n = 1;
    private int n_pri = 2;
    protected int n_pro = 3;
    public int n_pub = 4;

    public Protection() {
        System.out.println("base constructor");
        System.out.println("n = " + n);
        System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

This is file **Derived.java**:

```
package p1;

class Derived extends Protection {
    Derived() {
        System.out.println("derived constructor");
        System.out.println("n = " + n);

        // class only
        // System.out.println("n_pri = " + n_pri);

        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

This is file **SamePackage.java**:

```
package p1;

class SamePackage {
    SamePackage() {

        Protection p = new Protection();
        System.out.println("same package constructor");
        System.out.println("n = " + p.n);

        // class only
        // System.out.println("n_pri = " + p.n_pri);

        System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}
```

Following is the source code for the other package, **p2**. The two classes defined in **p2** cover the other two conditions that are affected by access control. The first class, **Protection2**, is a subclass of **p1.Protection**. This grants access to all of **p1.Protection**'s variables except for **n_pri** (because it is **private**) and **n**, the variable declared with the default protection. Remember, the default only allows access from within the class or the

package, not extra-package subclasses. Finally, the class **OtherPackage** has access to only one variable, **n_pub**, which was declared **public**.

This is file **Protection2.java**:

```
package p2;

class Protection2 extends p1.Protection {
    Protection2() {
        System.out.println("derived other package constructor");

        // class or package only
        // System.out.println("n = " + n);

        // class only
        // System.out.println("n_pri = " + n_pri);

        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

This is file **OtherPackage.java**:

```
package p2;

class OtherPackage {
    OtherPackage() {
        p1.Protection p = new p1.Protection();
        System.out.println("other package constructor");

        // class or package only
        // System.out.println("n = " + p.n);

        // class only
        // System.out.println("n_pri = " + p.n_pri);

        // class, subclass or package only
        // System.out.println("n_pro = " + p.n_pro);

        System.out.println("n_pub = " + p.n_pub);
    }
}
```

If you want to try these two packages, here are two test files you can use. The one for package **p1** is shown here:

```
// Demo package p1.
package p1;

// Instantiate the various classes in p1.
public class Demo {
    public static void main(String[] args) {
        Protection ob1 = new Protection();
        Derived ob2 = new Derived();
        SamePackage ob3 = new SamePackage();
    }
}
```

The test file for **p2** is shown next:

```
// Demo package p2.  
package p2;  
  
// Instantiate the various classes in p2.  
public class Demo {  
    public static void main(String[] args) {  
        Protection2 ob1 = new Protection2();  
        OtherPackage ob2 = new OtherPackage();  
    }  
}
```

Importing Packages

Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages. There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use. For this reason, Java includes the **import** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.

In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. This is the general form of the **import** statement:

```
import pkg1 [pkg2].(classname | *);
```

Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit *classname* or a star (*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use:

```
import java.util.Date;  
import java.io.*;
```

All of the standard Java SE classes included with Java begin with the name **java**. The basic language functions are stored in a package called **java.lang**. Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in **java.lang**, it is implicitly imported by the compiler for all programs. This is equivalent to the following line being at the top of all of your programs:

```
import java.lang.*;
```

If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile-time error and have to explicitly name the class specifying its package.

It must be emphasized that the **import** statement is optional. Any place you use a class name, you can use its *fully qualified name*, which includes its full package hierarchy. For example, this fragment uses an **import** statement:

```
import java.util.*;  
class MyDate extends Date {  
}
```

The same example without the **import** statement looks like this:

```
class MyDate extends java.util.Date {  
}
```

In this version, **Date** is fully qualified.

As shown in [Table 9-1](#), when a package is imported, only those items within the package declared as **public** will be available to non-subclasses in the importing code. For example, if you want the **Balance** class of the package **mypad** shown earlier to be available as a stand-alone class for general use outside of **mypad**, then you will need to declare it as **public** and put it into its own file, as shown here:

```
package mypack;

/* Now, the Balance class, its constructor, and its
   show() method are public. This means that they can
   be used by non-subclass code outside their package.
*/
public class Balance {
    String name;
    double bal;

    public Balance(String n, double b) {
        name = n;
        bal = b;
    }

    public void show() {
        if(bal<0)

            System.out.print("--> ");
            System.out.println(name + ": $" + bal);
    }
}
```

As you can see, the **Balance** class is now **public**. Also, its constructor and its **show()** method are **public**, too. This means that they can be accessed by any type of code outside the **mypack** package. For example, here **TestBalance** imports **mypack** and is then able to make use of the **Balance** class:

```
import mypack.*;

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

        /* Because Balance is public, you may use Balance
           class and call its constructor. */
        Balance test = new Balance("J. J. Jaspers", 99.88);

        test.show(); // you may also call show()
    }
}
```

As an experiment, remove the **public** specifier from the **Balance** class and then try compiling **TestBalance**. As explained, errors will result.

Interfaces

Using the keyword **interface**, you can fully abstract a class's interface from its implementation. That is, using **interface**, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body. In practice, this means that you can define interfaces that don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.

To implement an interface, a class must provide the complete set of methods required by the interface. However, each class is free to determine the details of its own implementation. By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and nonextensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

Defining an Interface

An interface is defined much like a class. This is a simplified general form of an interface:

```

access interface name {
    return-type method-name1(parameter-list);
    return-type method-name2(parameter-list);

    type final-varname1 = value;
    type final-varname2 = value;
    //...
    return-type method-nameN(parameter-list);
    type final-varnameN = value;
}

```

When no access modifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by code outside its package. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface. *name* is the name of the interface, and can be any valid identifier. Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods. Each class that includes such an interface must implement all of the methods.

Before continuing, an important point needs to be made. JDK 8 added a feature to **interface** that made a significant change to its capabilities. Prior to JDK 8, an interface could not define any implementation whatsoever. This is the type of interface that the preceding simplified form shows, in which no method declaration supplies a body. Thus, prior to JDK 8, an interface could define only “what,” but not “how.” JDK 8 changed this. Beginning with JDK 8, it is possible to add a *default implementation* to an interface method. Furthermore, JDK 8 also added static interface methods, and beginning with JDK 9, an interface can include private methods. Thus, it is now possible for **interface** to specify some behavior. However, such methods constitute what are, in essence, special-use features, and the original intent behind **interface** still remains. Therefore, as a general rule, you will still often create and use interfaces in which no use is made of these new features. For this reason, we will begin by discussing the

interface in its traditional form. The newer interface features are described at the end of this chapter.

As the general form shows, variables can be declared inside interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly **public**.

Here is an example of an interface definition. It declares a simple interface that contains one method called **callback()** that takes a single integer parameter.

```
interface Callback {  
    void callback(int param);  
}
```

Implementing Interfaces

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the **implements** clause in a class definition, and then create the methods required by the interface. The general form of a class that includes the **implements** clause looks like this:

```
class classname [extends superclass] [implements interface [,interface...]] {  
    // class-body  
}
```

If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared **public**. Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

Here is a small sample class that implements the **Callback** interface shown earlier:

```
class Client implements Callback {  
    // Implement Callback's interface  
    public void callback(int p) {  
  
        System.out.println("callback called with " + p);  
    }  
}
```

Notice that **callback()** is declared using the **public** access modifier.

REMEMBER When you implement an interface method, it must be declared as **public**.

It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of **Client** implements **callback()** and adds the method **nonIfaceMeth()**:

```
class Client implements Callback {  
    // Implement Callback's interface  
    public void callback(int p) {  
        System.out.println("callback called with " + p);  
    }  
  
    void nonIfaceMeth() {  
        System.out.println("Classes that implement interfaces " +  
            "may also define other members, too.");  
    }  
}
```

Accessing Implementations Through Interface References

You can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code that calls methods on them. The calling code can dispatch through an interface without having to know anything about the “callee.” This process is similar

to using a superclass reference to access a subclass object, as described in [Chapter 8](#).

The following example calls the **callback()** method via an interface reference variable:

```
class TestIface {
    public static void main(String[] args) {
        Callback c = new Client();
        c.callback(42);
    }
}
```

The output of this program is shown here:

```
callback called with 42
```

Notice that variable **c** is declared to be of the interface type **Callback**, yet it was assigned an instance of **Client**. Although **c** can be used to access the **callback()** method, it cannot access any other members of the **Client** class. An interface reference variable has knowledge only of the methods declared by its **interface** declaration. Thus, **c** could not be used to access **nonIfaceMeth()** since it is defined by **Client** but not **Callback**.

While the preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference. To sample this usage, first create the second implementation of **Callback**, shown here:

```
// Another implementation of Callback.
class AnotherClient implements Callback {
    // Implement Callback's interface
    public void callback(int p) {
        System.out.println("Another version of callback");
        System.out.println("p squared is " + (p*p));
    }
}
```

Now, try the following class:

```
class TestIface2 {  
    public static void main(String[] args) {  
        Callback c = new Client();  
        AnotherClient ob = new AnotherClient();  
  
        c.callback(42);  
  
        c = ob; // c now refers to AnotherClient object  
        c.callback(42);  
    }  
}
```

The output from this program is shown here:

```
callback called with 42  
Another version of callback  
p squared is 1764
```

As you can see, the version of **callback()** that is called is determined by the type of object that **c** refers to at run time. While this is a very simple example, you will see another, more practical one shortly.

Partial Implementations

If a class includes an interface but does not fully implement the methods required by that interface, then that class must be declared as **abstract**. For example:

```
abstract class Incomplete implements Callback {  
    int a, b;  
  
    void show() {  
        System.out.println(a + " " + b);  
    }  
    //...  
}
```

Here, the class **Incomplete** does not implement **callback()** and must be declared as **abstract**. Any class that inherits **Incomplete** must implement **callback()** or be declared **abstract** itself.

Nested Interfaces

An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*. A nested interface can be declared as **public**, **private**, or **protected**. This differs from a top-level interface, which must either be declared as **public** or use the default access level, as previously described. When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member. Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

Here is an example that demonstrates a nested interface:

```
// A nested interface example.

// This class contains a member interface.
class A {
    // this is a nested interface
    public interface NestedIF {
        boolean isNotNegative(int x);
    }
}
```

```

// B implements the nested interface.
class B implements A.NestedIF {
    public boolean isNotNegative(int x) {
        return x < 0 ? false: true;
    }
}

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

        // use a nested interface reference
        A.NestedIF nif = new B();

        if(nif.isNotNegative(10))
            System.out.println("10 is not negative");
        if(nif.isNotNegative(-12))
            System.out.println("this won't be displayed");
    }
}

```

Notice that **A** defines a member interface called **NestedIF** and that it is declared **public**. Next, **B** implements the nested interface by specifying

implements A.NestedIF

Notice that the name is fully qualified by the enclosing class's name. Inside the **main()** method, an **A.NestedIF** reference called **nif** is created, and it is assigned a reference to a **B** object. Because **B** implements **A.NestedIF**, this is legal.

Applying Interfaces

To understand the power of interfaces, let's look at a more practical example. In earlier chapters, you developed a class called **Stack** that implemented a simple fixed-size stack. However, there are many ways to implement a stack. For example, the stack can be of a fixed size or it can be "growable." The stack can also be held in an array, a linked list, a binary tree, and so on. No matter how the stack is implemented, the interface to the stack remains the same. That is, the methods **push()** and **pop()** define

the interface to the stack independently of the details of the implementation. Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics. Let's look at two examples.

First, here is the interface that defines an integer stack. Put this in a file called **IntStack.java**. This interface will be used by both stack implementations.

```
// Define an integer stack interface.  
interface IntStack {  
    void push(int item); // store an item  
    int pop(); // retrieve an item  
}
```

The following program creates a class called **FixedStack** that implements a fixed-length version of an integer stack:

```
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
    private int[] stck;
    private int tos;

    // allocate and initialize stack
    FixedStack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // Push an item onto the stack
    public void push(int item) {
        if(tos==stck.length-1) // use length member
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }

    // Pop an item from the stack
    public int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}

class IFTest {
    public static void main(String[] args) {
        FixedStack mystack1 = new FixedStack(5);
        FixedStack mystack2 = new FixedStack(8);

        // push some numbers onto the stack
        for(int i=0; i<5; i++) mystack1.push(i);
        for(int i=0; i<8; i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());
    }
}
```

Following is another implementation of **IntStack** that creates a dynamic stack by use of the same **interface** definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled.

```
System.out.println(mystack1.pop());  
System.out.println(mystack1.pop());  
System.out.println(mystack2.pop());  
}  
}
```

```
// Implement a "growable" stack.
class DynStack implements IntStack {
    private int[] stck;
    private int tos;

    // allocate and initialize stack
    DynStack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // Push an item onto the stack
    public void push(int item) {
        // if stack is full, allocate a larger stack
        if(tos==stck.length-1) {
            int[] temp = new int[stck.length * 2]; // double size
            for(int i=0; i<stck.length; i++) temp[i] = stck[i];
            stck = temp;
            stck[++tos] = item;
        }
        else
            stck[++tos] = item;
    }

    // Pop an item from the stack
    public int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}

class IFTest2 {
    public static void main(String[] args) {
        DynStack mystack1 = new DynStack(5);
        DynStack mystack2 = new DynStack(8);

        // these loops cause each stack to grow
        for(int i=0; i<12; i++) mystack1.push(i);
    }
}
```

```
        System.out.println("Stack in mystack2:");
        for(int i=0; i<20; i++)
            System.out.println(mystack2.pop());
    }
}
System.out.println(mystack1.pop());
```

The following class uses both the **FixedStack** and **DynStack** implementations. It does so through an interface reference. This means that calls to **push()** and **pop()** are resolved at run time rather than at compile time.

```
/* Create an interface variable and
   access stacks through it.
*/
class IFTest3 {
    public static void main(String[] args) {
        IntStack mystack; // create an interface reference variable
        DynStack ds = new DynStack(5);
        FixedStack fs = new FixedStack(8);

        mystack = ds; // load dynamic stack
        // push some numbers onto the stack
        for(int i=0; i<12; i++) mystack.push(i);

        mystack = fs; // load fixed stack
        for(int i=0; i<8; i++) mystack.push(i);

        mystack = ds;
        System.out.println("Values in dynamic stack:");
        for(int i=0; i<12; i++)
            System.out.println(mystack.pop());

        mystack = fs;
        System.out.println("Values in fixed stack:");
        for(int i=0; i<8; i++)
            System.out.println(mystack.pop());
    }
}
```

In this program, **mystack** is a reference to the **IntStack** interface. Thus, when it refers to **ds**, it uses the versions of **push()** and **pop()** defined by the **DynStack** implementation. When it refers to **fs**, it uses the versions of **push()** and **pop()** defined by **FixedStack**. As explained, these determinations are made at run time. Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

Variables in Interfaces

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When you include that interface in a class (that is, when you “implement” the interface), all of those variable names will be in scope as constants. If an interface contains no methods, then any class that includes such an interface doesn’t actually implement anything. It is as if that class were importing the constant fields into the class name space as **final** variables. The next example uses this technique to implement an automated “decision maker”:

```
import java.util.Random;

interface SharedConstants {
    int NO = 0;
    int YES = 1;
    int MAYBE = 2;
    int LATER = 3;
    int SOON = 4;
    int NEVER = 5;
}

class Question implements SharedConstants {
    Random rand = new Random();
    int ask() {
        int prob = (int) (100 * rand.nextDouble());
        if (prob < 30)
            return NO;                  // 30%
        else if (prob < 60)
            return YES;                // 30%
        else if (prob < 75)
            return LATER;              // 15%
        else if (prob < 98)
            return SOON;               // 13%
        else
            return NEVER;             // 2%
    }
}

class AskMe implements SharedConstants {
    static void answer(int result) {
        switch(result) {
            case NO:
                System.out.println("No");
                break;
            case YES:
                System.out.println("Yes");
                break;
            case MAYBE:
                System.out.println("Maybe");
                break;
            case LATER:
                System.out.println("Later");
                break;
        }
    }
}
```

```

    }
}

public static void main(String[] args) {
    Question q = new Question();

    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
}
}

```

Notice that this program makes use of one of Java's standard classes:

Random. This class provides pseudorandom numbers. It contains several methods that allow you to obtain random numbers in the form required by your program. In this example, the method **nextDouble()** is used. It returns random numbers in the range 0.0 to 1.0.

In this sample program, the two classes, **Question** and **AskMe**, both implement the **SharedConstants** interface where **NO**, **YES**, **MAYBE**, **SOON**, **LATER**, and **NEVER** are defined. Inside each class, the code refers to these constants as if each class had defined or inherited them directly. Here is the output of a sample run of this program. Note that the results are different each time it is run.

```

Later
Soon
No
Yes

```

NOTE The technique of using an interface to define shared constants, as just described, is controversial. It is described here for completeness.

Interfaces Can Be Extended

One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for

all methods required by the interface inheritance chain. Following is an example:

```
// One interface can extend another.
interface A {
    void meth1();
    void meth2();
}

// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
    void meth3();
}

// This class must implement all of A and B
class MyClass implements B {
    public void meth1() {
        System.out.println("Implement meth1().");
    }

    public void meth2() {
        System.out.println("Implement meth2().");
    }

    public void meth3() {
        System.out.println("Implement meth3().");
    }
}

class IFExtend {
    public static void main(String[] args) {
        MyClass ob = new MyClass();

        ob.meth1();
        ob.meth2();
        ob.meth3();
    }
}
```

As an experiment, you might want to try removing the implementation for **meth1()** in **MyClass**. This will cause a compile-time error. As stated

earlier, any class that implements an interface must implement all methods required by that interface, including any that are inherited from other interfaces.

Default Interface Methods

As explained earlier, prior to JDK 8, an interface could not define any implementation whatsoever. This meant that for all previous versions of Java, the methods specified by an interface were abstract, containing no body. This is the traditional form of an interface and is the type of interface that the preceding discussions have used. The release of JDK 8 changed this by adding a new capability to **interface** called the *default method*. A default method lets you define a default implementation for an interface method. In other words, by use of a default method, it is possible for an interface method to provide a body, rather than being abstract. During its development, the default method was also referred to as an *extension method*, and you will likely see both terms used.

A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code. Recall that there must be implementations for all methods defined by an interface. In the past, if a new method were added to a popular, widely used interface, then the addition of that method would break existing code because no implementation would be found for that new method. The default method solves this problem by supplying an implementation that will be used if no other implementation is explicitly provided. Thus, the addition of a default method will not cause preexisting code to break.

Another motivation for the default method was the desire to specify methods in an interface that are, essentially, optional, depending on how the interface is used. For example, an interface might define a group of methods that act on a sequence of elements. One of these methods might be called **remove()**, and its purpose is to remove an element from the sequence. However, if the interface is intended to support both modifiable and nonmodifiable sequences, then **remove()** is essentially optional because it won't be used by nonmodifiable sequences. In the past, a class that implemented a nonmodifiable sequence would have had to define an empty implementation of **remove()**, even though it was not needed. Today, a default implementation for **remove()** can be specified in the interface that

does nothing (or throws an exception). Providing this default prevents a class used for nonmodifiable sequences from having to define its own, placeholder version of **remove()**. Thus, by providing a default, the interface makes the implementation of **remove()** by a class optional.

It is important to point out that the addition of default methods does not change a key aspect of **interface**: its inability to maintain state information. An interface still cannot have instance variables, for example. Thus, the defining difference between an interface and a class is that a class can maintain state information, but an interface cannot. Furthermore, it is still not possible to create an instance of an interface by itself. It must be implemented by a class. Therefore, even though, beginning with JDK 8, an interface can define default methods, the interface must still be implemented by a class if an instance is to be created.

One last point: As a general rule, default methods constitute a special-purpose feature. Interfaces that you create will still be used primarily to specify *what* and not *how*. However, the inclusion of the default method gives you added flexibility.

Default Method Fundamentals

An interface default method is defined similar to the way a method is defined by a **class**. The primary difference is that the declaration is preceded by the keyword **default**. For example, consider this simple interface:

```
public interface MyIF {  
    // This is a "normal" interface method declaration.  
    // It does NOT define a default implementation.  
    int getNumber();  
  
    // This is a default method. Notice that it provides  
    // a default implementation.  
    default String getString() {  
        return "Default String";  
    }  
}
```

MyIF declares two methods. The first, **getNumber()**, is a standard interface method declaration. It defines no implementation whatsoever. The second method is **getString()**, and it does include a default implementation. In this case, it simply returns the string "Default String". Pay special attention to the way **getString()** is declared. Its declaration is preceded by the **default** modifier. This syntax can be generalized. To define a default method, precede its declaration with **default**.

Because **getString()** includes a default implementation, it is not necessary for an implementing class to override it. In other words, if an implementing class does not provide its own implementation, the default is used. For example, the **MyIFImp** class shown next is perfectly valid:

```
// Implement MyIF.
class MyIFImp implements MyIF {
    // Only getNumber() defined by MyIF needs to be implemented.
    // getString() can be allowed to default.
    public int getNumber() {
        return 100;
    }
}
```

The following code creates an instance of **MyIFImp** and uses it to call both **getNumber()** and **getString()**:

```
// Use the default method.
class DefaultMethodDemo {
    public static void main(String[] args) {

        MyIFImp obj = new MyIFImp();

        // Can call getNumber(), because it is explicitly
        // implemented by MyIFImp:
        System.out.println(obj.getNumber());

        // Can also call getString(), because of default
        // implementation:
        System.out.println(obj.getString());
    }
}
```

The output is shown here:

```
100
Default String
```

As you can see, the default implementation of **getString()** was automatically used. It was not necessary for **MyIFImp** to define it. Thus, for **getString()**, implementation by a class is optional. (Of course, its implementation by a class will be *required* if the class uses **getString()** for some purpose beyond that supported by its default.)

It is both possible and common for an implementing class to define its own implementation of a default method. For example, **MyIFImp2** overrides **getString()**:

```
class MyIFImp2 implements MyIF {
    // Here, implementations for both getNumber( ) and getString( ) are provided.
    public int getNumber() {
        return 100;
    }

    public String getString() {
        return "This is a different string.";
    }
}
```

Now, when **getString()** is called, a different string is returned.

A More Practical Example

Although the preceding shows the mechanics of using default methods, it doesn't illustrate their usefulness in a more practical setting. To do this, let's once again return to the **IntStack** interface shown earlier in this chapter. For the sake of discussion, assume that **IntStack** is widely used and many programs rely on it. Further assume that we now want to add a method to **IntStack** that clears the stack, enabling the stack to be re-used. Thus, we want to evolve the **IntStack** interface so that it defines new functionality, but we don't want to break any preexisting code. In the past, this would be impossible, but with the inclusion of default methods, it is

now easy to do. For example, the **IntStack** interface can be enhanced like this:

```
interface IntStack {  
    void push(int item); // store an item  
    int pop(); // retrieve an item  
  
    // Because clear( ) has a default, it need not be  
    // implemented by a preexisting class that uses IntStack.  
    default void clear() {  
        System.out.println("clear() not implemented.");  
    }  
}
```

Here, the default behavior of **clear()** simply displays a message indicating that it is not implemented. This is acceptable because no preexisting class that implements **IntStack** would ever call **clear()** because it was not defined by the earlier version of **IntStack**. However, **clear()** can be implemented by a new class that implements **IntStack**. Furthermore, **clear()** needs to be defined by a new implementation only if it is used. Thus, the default method gives you

- a way to gracefully evolve interfaces over time, and
- a way to provide optional functionality without requiring that a class provide a placeholder implementation when that functionality is not needed.

One other point: In real-world code, **clear()** would have thrown an exception, rather than displaying an error message. Exceptions are described in the next chapter. After working through that material, you might want to try modifying **clear()** so that its default implementation throws an **UnsupportedOperationException**.

Multiple Inheritance Issues

As explained earlier in this book, Java does not support the multiple inheritance of classes. Now that an interface can include default methods, you might be wondering if an interface can provide a way around this

restriction. The answer is, essentially, no. Recall that there is still a key difference between a class and an interface: a class can maintain state information (especially through the use of instance variables), but an interface cannot.

The preceding notwithstanding, default methods do offer a bit of what one would normally associate with the concept of multiple inheritance. For example, you might have a class that implements two interfaces. If each of these interfaces provides default methods, then some behavior is inherited from both. Thus, to a limited extent, default methods do support multiple inheritance of behavior. As you might guess, in such a situation, it is possible that a name conflict will occur.

For example, assume that two interfaces called **Alpha** and **Beta** are implemented by a class called **MyClass**. What happens if both **Alpha** and **Beta** provide a method called **reset()** for which both declare a default implementation? Is the version by **Alpha** or the version by **Beta** used by **MyClass**? Or, consider a situation in which **Beta** extends **Alpha**. Which version of the default method is used? Or, what if **MyClass** provides its own implementation of the method? To handle these and other similar types of situations, Java defines a set of rules that resolves such conflicts.

First, in all cases, a class implementation takes priority over an interface default implementation. Thus, if **MyClass** provides an override of the **reset()** default method, **MyClass**'s version is used. This is the case even if **MyClass** implements both **Alpha** and **Beta**. In this case, both defaults are overridden by **MyClass**'s implementation.

Second, in cases in which a class implements two interfaces that both have the same default method, but the class does not override that method, then an error will result. Continuing with the example, if **MyClass** implements both **Alpha** and **Beta**, but does not override **reset()**, then an error will occur.

In cases in which one interface inherits another, with both defining a common default method, the inheriting interface's version of the method takes precedence. Therefore, continuing the example, if **Beta** extends **Alpha**, then **Beta**'s version of **reset()** will be used.

It is possible to explicitly refer to a default implementation in an inherited interface by using this form of **super**. Its general form is shown here:

InterfaceName.super.methodName()

For example, if **Beta** wants to refer to **Alpha**'s default for **reset()**, it can use this statement:

```
Alpha.super.reset();
```

Use static Methods in an Interface

Another capability added to **interface** by JDK 8 is the ability to define one or more **static** methods. Like **static** methods in a class, a **static** method defined by an interface can be called independently of any object. Thus, no implementation of the interface is necessary, and no instance of the interface is required, in order to call a **static** method. Instead, a **static** method is called by specifying the interface name, followed by a period, followed by the method name. Here is the general form:

InterfaceName.staticMethodName

Notice that this is similar to the way that a **static** method in a class is called.

The following shows an example of a **static** method in an interface by adding one to **MyIF**, shown in the previous section. The **static** method is **getDefaultNumber()**. It returns zero.

```
public interface MyIF {
    // This is a "normal" interface method declaration.
    // It does NOT define a default implementation.
    int getNumber();

    // This is a default method. Notice that it provides
    // a default implementation.
    default String getString() {
        return "Default String";
    }

    // This is a static interface method.
    static int getDefaultNumber() {
        return 0;
    }
}
```

The **getDefaultNumber()** method can be called as shown here:

```
int defNum = MyIF.getDefaultNumber();
```

As mentioned, no implementation or instance of **MyIF** is required to call **getDefaultNumber()** because it is **static**.

One last point: **static** interface methods are not inherited by either an implementing class or a subinterface.

Private Interface Methods

Beginning with JDK 9, an interface can include a private method. A private interface method can be called only by a default method or another private method defined by the same interface. Because a private interface method is specified **private**, it cannot be used by code outside the interface in which it is defined. This restriction includes subinterfaces because a private interface method is not inherited by a subinterface.

The key benefit of a private interface method is that it lets two or more default methods use a common piece of code, thus avoiding code duplication. For example, here is another version of the **IntStack** interface that has two default methods called **popNElements()** and **skipAndPopNElements()**. The first returns an array that contains the top

N elements on the stack. The second skips a specified number of elements and then returns an array that contains the next N elements. Both use a private method called **getElements()** to obtain an array of the specified number of elements from the stack.

```
// Another version of IntStack that has a private interface
// method that is used by two default methods.
interface IntStack {
    void push(int item); // store an item
    int pop(); // retrieve an item

    // A default method that returns an array that contains
    // the top n elements on the stack.
    default int[] popNElements(int n) {
        // Return the requested elements.
        return getElements(n);
    }

    // A default method that returns an array that contains
    // the next n elements on the stack after skipping elements.
    default int[] skipAndPopNElements(int skip, int n) {

        // Skip the specified number of elements.
        getElements(skip);

        // Return the requested elements.
        return getElements(n);
    }

    // A private method that returns an array containing
    // the top n elements on the stack
    private int[] getElements(int n) {
        int[] elements = new int[n];

        for(int i=0; i < n; i++) elements[i] = pop();
        return elements;
    }
}
```

Notice that both `popNElements()` and `skipAndPopNElements()` use the private `getElements()` method to obtain the array to return. This prevents both methods from having to duplicate the same code sequence. Keep in mind that because `getElements()` is private, it cannot be called by code outside `IntStack`. Thus, its use is limited to the default methods inside `IntStack`. Also, because `getElements()` uses the `pop()` method to obtain stack elements, it will automatically call the implementation of `pop()` provided by the `IntStack` implementation. Thus, `getElements()` will work for any stack class that implements `IntStack`.

Although the private interface method is a feature that you will seldom need, in those cases in which you *do* need it, you will find it quite useful.

Final Thoughts on Packages and Interfaces

Although the examples we've included in this book do not make frequent use of packages or interfaces, both of these tools are an important part of the Java programming environment. Virtually all real programs that you write in Java will be contained within packages. A number will probably implement interfaces as well. It is important, therefore, that you be comfortable with their usage.

CHAPTER

10

Exception Handling

This chapter examines Java’s exception-handling mechanism. An *exception* is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on. This approach is as cumbersome as it is troublesome. Java’s exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world.

Exception-Handling Fundamentals

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a

try block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

This is the general form of an exception-handling block:

```
try {  
    // block of code to monitor for errors  
}  
  
catch (ExceptionType1 exOb) {  
    // exception handler for ExceptionType1  
}  
  
catch (ExceptionType2 exOb) {  
    // exception handler for ExceptionType2  
}  
// ...  
finally {  
    // block of code to be executed after try block ends  
}
```

Here, *ExceptionType* is the type of exception that has occurred. The remainder of this chapter describes how to apply this framework.

NOTE There is another form of the **try** statement that supports *automatic resource management*. This form of **try**, called **try-with-resources**, is described in [Chapter 13](#) in the context of managing files because files are some of the most commonly used resources.

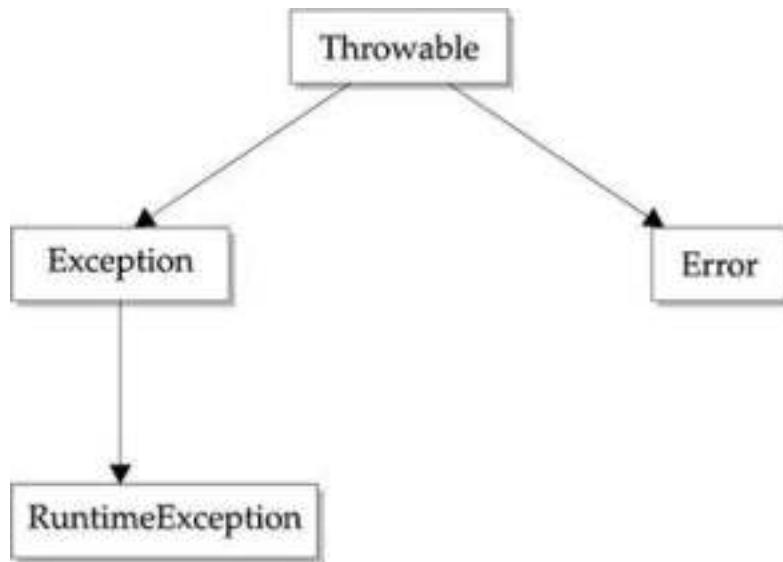
Exception Types

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy. Immediately below **Throwable** are two subclasses that partition exceptions into two

distinct branches. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment itself. Stack overflow is an example of such an error. This chapter will not be dealing with exceptions of type **Error**, because these are typically created in response to catastrophic failures that cannot usually be handled by your program.

The top-level exception hierarchy is shown here:



Uncaught Exceptions

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them. This small program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0 {  
    public static void main(String[] args) {  
        int d = 0;  
        int a = 42 / d;  
    }  
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the exception generated when this example is executed:

```
java.lang.ArithmetricException: / by zero  
at Exc0.main(Exc0.java:4)
```

Notice how the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, 4, are all included in the simple stack trace. Also, notice that the type of exception thrown is a subclass of **Exception** called **ArithmetricException**, which more specifically describes what type of error happened. As discussed later in this chapter, Java supplies several built-in exception types that match the various sorts of run-time errors that can be generated. One other point: The exact output you see when running this and other sample programs in this chapter that use Java's built-in exceptions may vary slightly from what is shown because of differences between JDks.

The stack trace will always show the sequence of method invocations that led up to the error. For example, here is another version of the preceding program that introduces the same error but in a method separate from **main()**:

```
class Excl {
    static void subroutine() {
        int d = 0;
        int a = 10 / d;
    }
    public static void main(String[] args) {
        Excl.subroutine();
    }
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmetricException: / by zero
at Excl.subroutine(Excl.java:4)
at Excl.main(Excl.java:7)
```

As you can see, the bottom of the stack is **main**'s line 7, which is the call to **subroutine()**, which caused the exception at line 4. The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the error.

Using try and catch

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating. Most users would be confused (to say the least) if your program stopped running and printed a stack trace whenever an error occurred! Fortunately, it is quite easy to prevent this.

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block. Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a **try** block and a **catch** clause that processes the **ArithmetricException** generated by the division-by-zero error:

```
class Exc2 {
    public static void main(String[] args) {
        int d, a;

        try { // monitor a block of code.
            d = 0;
            a = 42 / d;
            System.out.println("This will not be printed.");
        } catch (ArithmaticException e) { // catch divide-by-zero error
            System.out.println("Division by zero.");
        }

        System.out.println("After catch statement.");
    }
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

Notice that the call to **println()** inside the **try** block is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. Put differently, **catch** is not “called,” so execution never “returns” to the **try** block from a **catch**. Thus, the line “This will not be printed.” is not displayed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism.

A **try** and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement. A **catch** statement cannot catch an exception thrown by another **try** statement (except in the case of nested **try** statements, described shortly). The statements that are protected by **try** must be surrounded by curly braces. (That is, they must be within a block.) You cannot use **try** on a single statement.

The goal of most well-constructed **catch** clauses should be to resolve the exceptional condition and then continue on as if the error had never happened. For example, in the next program, each iteration of the **for** loop obtains two random integers. Those two integers are divided by each other,

and the result is used to divide the value 12345. The final result is put into **a**. If either division operation causes a divide-by-zero error, it is caught, the value of **a** is set to zero, and the program continues.

```
// Handle an exception and move on.  
import java.util.Random;  
  
class HandleError {  
    public static void main(String[] args) {  
        int a=0, b=0, c=0;  
        Random r = new Random();  
  
        for(int i=0; i<32000; i++) {  
            try {  
                b = r.nextInt();  
                c = r.nextInt();  
                a = 12345 / (b/c);  
            } catch (ArithmetricException e) {  
                System.out.println("Division by zero.");  
                a = 0; // set a to zero and continue  
            }  
            System.out.println("a: " + a);  
        }  
    }  
}
```

Displaying a Description of an Exception

Throwable overrides the **toString()** method (defined by **Object**) so that it returns a string containing a description of the exception. You can display this description in a **println()** statement by simply passing the exception as an argument. For example, the **catch** block in the preceding program can be rewritten like this:

```
catch (ArithmetricException e) {  
    System.out.println("Exception: " + e);  
    a = 0; // set a to zero and continue  
}
```

When this version is substituted in the program, and the program is run, each divide-by-zero error displays the following message:

```
Exception: java.lang.ArithmetricException: / by zero
```

While it is of no particular value in this context, the ability to display a description of an exception is valuable in other circumstances—particularly when you are experimenting with exceptions or when you are debugging.

Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try/catch** block. The following example traps two different exception types:

```
// Demonstrate multiple catch statements.
class MultipleCatches {
    public static void main(String[] args) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int[] c = { 1 };
            c[42] = 99;
        } catch(ArithmetricException e) {
            System.out.println("Divide by 0: " + e);
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index oob: " + e);
        }
        System.out.println("After try/catch blocks.");
    }
}
```

This program will cause a division-by-zero exception if it is started with no command-line arguments, since **a** will equal zero. It will survive the division if you provide a command-line argument, setting **a** to something larger than zero. But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.

Here is the output generated by running it both ways:

```
C:\>java MultipleCatches
a = 0
Divide by 0: java.lang.ArithmetricException: / by zero
After try/catch blocks.

C:\>java MultipleCatches TestArg
a = 1
Array index oob:  java.lang.ArrayIndexOutOfBoundsException:
Index 42 out of bounds for length 1
After try/catch blocks.
```

When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error. For example, consider the following program:

```
/* This program contains an error.

A subclass must come before its superclass in
a series of catch statements. If not,
unreachable code will be created and a
compile-time error will result.

*/
class SuperSubCatch {
    public static void main(String[] args) {
        try {
            int a = 0;
            int b = 42 / a;
        } catch(Exception e) {
            System.out.println("Generic Exception catch.");
        }
        /* This catch is never reached because
           ArithmeticException is a subclass of Exception. */
        catch(ArithmeticException e) { // ERROR - unreachable
            System.out.println("This is never reached.");
        }
    }
}
```

If you try to compile this program, you will receive an error message stating that the second **catch** statement is unreachable because the exception has already been caught. Since **ArithmeticException** is a subclass of **Exception**, the first **catch** statement will handle all **Exception**-based errors, including **ArithmeticException**. This means that the second **catch** statement will never execute. To fix the problem, reverse the order of the **catch** statements.

Nested try Statements

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**. Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the

nested **try** statements are exhausted. If no **catch** statement matches, then the Java run-time system will handle the exception. Here is an example that uses nested **try** statements:

```
// An example of nested try statements.
class NestTry {
    public static void main(String[] args) {
        try {
            int a = args.length;

            /* If no command-line args are present,
               the following statement will generate
               a divide-by-zero exception. */
            int b = 42 / a;

            System.out.println("a = " + a);

            try { // nested try block
                /* If one command-line arg is used,
                   then a divide-by-zero exception
                   will be generated by the following code. */
                if(a==1) a = a/(a-a); // division by zero

                /* If two command-line args are used,
                   then generate an out-of-bounds exception. */
                if(a==2) {
                    int[] c = { 1 };
                    c[42] = 99; // generate an out-of-bounds exception
                }
            } catch(ArrayIndexOutOfBoundsException e) {
                System.out.println("Array index out-of-bounds: " + e);
            }

        } catch(ArithmetricException e) {
            System.out.println("Divide by 0: " + e);
        }
    }
}
```

As you can see, this program nests one **try** block within another. The program works as follows. When you execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer **try** block. Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested **try** block. Since the inner block does not catch this exception, it is passed on to the outer **try** block, where it is handled. If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner **try** block. Here are sample runs that illustrate each case:

```
C:\>java NestTry
Divide by 0: java.lang.ArithmetricException: / by zero

C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmetricException: / by zero

C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
    java.lang.ArrayIndexOutOfBoundsException:
        Index 42 out of bounds for length 1
```

Nesting of **try** statements can occur in less obvious ways when method calls are involved. For example, you can enclose a call to a method within a **try** block. Inside that method is another **try** statement. In this case, the **try** within the method is still nested inside the outer **try** block, which calls the method. Here is the previous program recoded so that the nested **try** block is moved inside the method **nesttry()**:

```
/* Try statements can be implicitly nested via
   calls to methods. */
class MethNestTry {
    static void nesttry(int a) {
        try { // nested try block
            /* If one command-line arg is used,
               then a divide-by-zero exception
               will be generated by the following code. */
            if(a==1) a = a/(a-a); // division by zero

            /* If two command-line args are used,
               then generate an out-of-bounds exception. */
            if(a==2) {
                int[] c = { 1 };
                c[42] = 99; // generate an out-of-bounds exception
            }
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out-of-bounds: " + e);
        }
    }

    public static void main(String[] args) {
        try {
            int a = args.length;

            /* If no command-line args are present,
               the following statement will generate
               a divide-by-zero exception. */
            int b = 42 / a;
            System.out.println("a = " + a);

            nesttry(a);
        } catch(ArithmaticException e) {
            System.out.println("Divide by 0: " + e);
        }
    }
}
```

The output of this program is identical to that of the preceding example.

throw

So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

```
throw ThrowableInstance;
```

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions. There are two ways you can obtain a **Throwable** object: using a parameter in a **catch** clause or creating one with the **new** operator.

The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
// Demonstrate throw.
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch(NullPointerException e) {
            System.out.println("Caught inside demoproc.");
            throw e; // rethrow the exception
        }
    }

    public static void main(String[] args) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            System.out.println("Recaught: " + e);
        }
    }
}
```

This program gets two chances to deal with the same error. First, `main()` sets up an exception context and then calls `demoproc()`. The `demoproc()` method then sets up another exception-handling context and immediately throws a new instance of `NullPointerException`, which is caught on the next line. The exception is then rethrown. Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

The program also illustrates how to create one of Java's standard exception objects. Pay close attention to this line:

```
throw new NullPointerException("demo");
```

Here, `new` is used to construct an instance of `NullPointerException`. Many of Java's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to `print()`

or `println()`. It can also be obtained by a call to `getMessage()`, which is defined by `Throwable`.

throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a `throws` clause in the method's declaration. A `throws` clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type `Error` or `RuntimeException`, or any of their subclasses. All other exceptions that a method can throw must be declared in the `throws` clause. If they are not, a compile-time error will result.

This is the general form of a method declaration that includes a `throws` clause:

```
type method-name(parameter-list) throws exception-list
{
    // body of method
}
```

Here, `exception-list` is a comma-separated list of the exceptions that a method can throw.

Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a `throws` clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.
class ThrowsDemo {
    static void throwOne() {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String[] args) {
        throwOne();
    }
}
```

To make this example compile, you need to make two changes. First, you need to declare that **throwOne()** throws **IllegalAccessException**. Second, **main()** must define a **try/catch** statement that catches this exception.

The corrected example is shown here:

```
// This is now correct.
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String[] args) {
        try {
            throwOne();
        } catch (IllegalAccessException e) {
            System.out.println("Caught " + e);
        }
    }
}
```

Here is the output generated by running this sample program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

finally

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address this contingency.

finally creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block. The **finally** block will execute whether or not an exception is thrown. If an

exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

Here is a sample program that shows three methods that exit in various ways, none without executing their **finally** clauses:

```
// Demonstrate finally.
class FinallyDemo {
    // Throw an exception out of the method.
    static void procA() {
        try {
            System.out.println("inside procA");
            throw new RuntimeException("demo");
        } finally {
            System.out.println("inside finally");
        }
    }
}
```

```
    } finally {
        System.out.println("procA's finally");
    }
}

// Return from within a try block.
static void procB() {
    try {
        System.out.println("inside procB");
        return;
    } finally {
        System.out.println("procB's finally");
    }
}

// Execute a try block normally.
static void procC() {
    try {
        System.out.println("inside procC");
    } finally {
        System.out.println("procC's finally");
    }
}

public static void main(String[] args) {
    try {
        procA();
    } catch (Exception e) {
        System.out.println("Exception caught");
    }

    procB();
    procC();
}
}
```

In this example, **procA()** prematurely breaks out of the **try** by throwing an exception. The **finally** clause is executed on the way out. **procB()**'s **try** statement is exited via a **return** statement. The **finally** clause is executed

before `procB()` returns. In `procC()`, the `try` statement executes normally, without error. However, the `finally` block is still executed.

REMEMBER If a `finally` block is associated with a `try`, the `finally` block will be executed upon conclusion of the `try`.

Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

Java's Built-in Exceptions

Inside the standard package `java.lang`, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type **RuntimeException**. As previously explained, these exceptions need not be included in any method's `throws` list. In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in `java.lang` are listed in [Table 10-1](#). [Table 10-2](#) lists those exceptions defined by `java.lang` that must be included in a method's `throws` list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions*. In addition to the exceptions in `java.lang`, Java defines several more that relate to its other standard packages.

Exception	Meaning
ArithmaticException	Arithmatic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out of bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalCallerException	A method cannot be legally executed by the calling code.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out of bounds.
LayerInstantiationException	A module layer cannot be created.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotFoundException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Table 10-1 Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

Table 10-2 Java’s Checked Exceptions Defined in `java.lang`

Creating Your Own Exception Subclasses

Although Java’s built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications. This is quite easy to do: just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**). Your subclasses don’t need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.

The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them. They are shown in [Table 10-3](#). You may also wish to override one or more of these methods in exception classes that you create.

Method	Description
final void addSuppressed(Throwable exc)	Adds <i>exc</i> to the list of suppressed exceptions associated with the invoking exception. Primarily for use by the try-with-resources statement.
Throwable fillInStackTrace()	Returns a Throwable object that contains a completed stack trace. This object can be rethrown.
Throwable getCause()	Returns the exception that underlies the current exception. If there is no underlying exception, null is returned.
String getLocalizedMessage()	Returns a localized description of the exception.
String getMessage()	Returns a description of the exception.
StackTraceElement[] getStackTrace()	Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement . The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The StackTraceElement class gives your program access to information about each element in the trace, such as its method name.
final Throwable[] getSuppressed()	Obtains the suppressed exceptions associated with the invoking exception and returns an array that contains the result. Suppressed exceptions are primarily generated by the try-with-resources statement.
Throwable initCause(Throwable causeExc)	Associates <i>causeExc</i> with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace()	Displays the stack trace.
void printStackTrace(PrintStream stream)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter stream)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement[] elements)	Sets the stack trace to the elements passed in <i>elements</i> . This method is for specialized applications, not normal use.

Table 10-3 The Methods Defined by **Throwable**

String toString()	Returns a String object containing a description of the exception. This method is called by println() when outputting a Throwable object.
Exception()	
Exception(String msg)	

The first form creates an exception that has no description. The second form lets you specify a description of the exception.

Although specifying a description when an exception is created is often useful, sometimes it is better to override **toString()**. Here's why: The version of **toString()** defined by **Throwable** (and inherited by **Exception**) first displays the name of the exception followed by a colon, which is then followed by your description. By overriding **toString()**, you can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.

The following example declares a new subclass of **Exception** and then uses that subclass to signal an error condition in a method. It overrides the **toString()** method, allowing a carefully tailored description of the exception to be displayed.

```
// This program creates a custom exception type.
class MyException extends Exception {
    private int detail;

    MyException(int a) {
        detail = a;
    }

    public String toString() {
        return "MyException[" + detail + "]";
    }
}

class ExceptionDemo {
    static void compute(int a) throws MyException {
        System.out.println("Called compute(" + a + ")");
        if(a > 10)
            throw new MyException(a);
        System.out.println("Normal exit");
    }
}

public static void main(String[] args) {
    try {
        compute(1);
        compute(20);
    } catch (MyException e) {
        System.out.println("Caught " + e);
    }
}
```

This example defines a subclass of **Exception** called **MyException**. This subclass is quite simple: It has only a constructor plus an overridden **toString()** method that displays the value of the exception. The **ExceptionDemo** class defines a method named **compute()** that throws a **MyException** object. The exception is thrown when **compute()**'s integer parameter is greater than 10. The **main()** method sets up an exception handler for **MyException**, then calls **compute()** with a legal value (less than 10) and an illegal one to show both paths through the code. Here is the result:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

Chained Exceptions

A number of years ago, a feature was incorporated into the exception subsystem: *chained exceptions*. The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception. For example, imagine a situation in which a method throws an **ArithmaticException** because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly. Although the method must certainly throw an **ArithmaticException**, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error. Chained exceptions let you handle this, and any other situation in which layers of exceptions exist.

To allow chained exceptions, two constructors and two methods were added to **Throwable**. The constructors are shown here:

```
Throwable(Throwable causeExc)
Throwable(String msg, Throwable causeExc)
```

In the first form, *causeExc* is the exception that causes the current exception. That is, *causeExc* is the underlying reason that an exception occurred. The second form allows you to specify a description at the same time that you specify a cause exception. These two constructors have also been added to the **Error**, **Exception**, and **RuntimeException** classes.

The chained exception methods supported by **Throwable** are **getCause()** and **initCause()**. These methods are shown in [Table 10-3](#) and are repeated here for the sake of discussion.

```
Throwable getCause()
Throwable initCause(Throwable causeExc)
```

The **getCause()** method returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned. The **initCause()** method associates *causeExc* with the invoking exception and returns a reference to the exception. Thus, you can associate a cause with an exception after the exception has been created. However, the cause exception can be set only once. This means that you can call **initCause()** only once for each exception object. Furthermore, if the cause exception was set by a constructor, then you can't set it again using **initCause()**. In general, **initCause()** is used to set a cause for legacy exception classes that don't support the two additional constructors described earlier.

Here is an example that illustrates the mechanics of handling chained exceptions:

```
// Demonstrate exception chaining.
class ChainExcDemo {
    static void demoproc() {

        // create an exception
        NullPointerException e =
            new NullPointerException("top layer");

        // add a cause
        e.initCause(new ArithmeticException("cause"));

        throw e;
    }

    public static void main(String[] args) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            // display top level exception
            System.out.println("Caught: " + e);

            // display cause exception
            System.out.println("Original cause: " +
                e.getCause());
        }
    }
}
```

The output from the program is shown here:

```
Caught: java.lang.NullPointerException: top layer
Original cause: java.lang.ArithmetricException: cause
```

In this example, the top-level exception is **NullPointerException**. To it is added a cause exception, **ArithmetricException**. When the exception is thrown out of **demoproc()**, it is caught by **main()**. There, the top-level exception is displayed, followed by the underlying exception, which is obtained by calling **getCause()**.

Chained exceptions can be carried on to whatever depth is necessary. Thus, the cause exception can, itself, have a cause. Be aware that overly long chains of exceptions may indicate poor design.

Chained exceptions are not something that every program will need. However, in cases in which knowledge of an underlying cause is useful, they offer an elegant solution.

Three Additional Exception Features

Beginning with JDK 7, three interesting and useful features have been part of the exception system. The first automates the process of releasing a resource, such as a file, when it is no longer needed. It is based on an expanded form of the **try** statement called **try-with-resources**, and is described in [Chapter 13](#) when files are introduced. The second feature is called *multi-catch*, and the third is sometimes referred to as *final rethrow* or *more precise rethrow*. These two features are described here.

The multi-catch feature allows two or more exceptions to be caught by the same **catch** clause. It is not uncommon for two or more exception handlers to use the same code sequence even though they respond to different exceptions. Instead of having to catch each exception type individually, you can use a single **catch** clause to handle all of the exceptions without code duplication.

To use a multi-catch, separate each exception type in the **catch** clause with the OR operator. Each multi-catch parameter is implicitly **final**. (You can explicitly specify **final**, if desired, but it is not necessary.) Because each multi-catch parameter is implicitly **final**, it can't be assigned a new value.

Here is a **catch** statement that uses the multi-catch feature to catch both **ArithmetiException** and **ArrayIndexOutOfBoundsException**:

```
catch(ArithmetiException | ArrayIndexOutOfBoundsException  
e) {
```

The following program shows the multi-catch feature in action:

```
// Demonstrate the multi-catch feature.  
class MultiCatch {  
    public static void main(String[] args) {  
        int a=10, b=0;  
        int[] vals = { 1, 2, 3 };  
  
        try {  
            int result = a / b; // generate an ArithmetiException  
  
//          vals[10] = 19; // generate an ArrayIndexOutOfBoundsException  
  
            // This catch clause catches both exceptions.  
        } catch(ArithmetiException | ArrayIndexOutOfBoundsException e) {  
            System.out.println("Exception caught: " + e);  
        }  
  
        System.out.println("After multi-catch.");  
    }  
}
```

The program will generate an **ArithmetiException** when the division by zero is attempted. If you comment out the division statement and remove the comment symbol from the next line, an **ArrayIndexOutOfBoundsException** is generated. Both exceptions are caught by the single **catch** statement.

The more precise rethrow feature restricts the type of exceptions that can be rethrown to only those checked exceptions that the associated **try** block throws, that are not handled by a preceding **catch** clause, and that are a subtype or supertype of the parameter. Although this capability might not be needed often, it is now available for use. For the more precise rethrow feature to be in force, the **catch** parameter must be either effectively **final**,

which means that it must not be assigned a new value inside the **catch** block, or explicitly declared **final**.

Using Exceptions

Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics. It is important to think of **try**, **throw**, and **catch** as clean ways to handle errors and unusual boundary conditions in your program's logic. Instead of using error return codes to indicate failure, use Java's exception handling capabilities. Thus, when a method can fail, have it throw an exception. This is a cleaner way to handle failure modes.

One last point: Java's exception-handling statements should not be considered a general mechanism for nonlocal branching. If you do so, it will only confuse your code and make it hard to maintain.

CHAPTER

11

Multithreaded Programming

Java provides built-in support for *multithreaded programming*. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread*, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

You are almost certainly acquainted with multitasking because it is supported by virtually all modern operating systems. However, there are two distinct types of multitasking: process-based and thread-based. It is important to understand the difference between the two. For many readers, process-based multitasking is the more familiar form. A *process* is, in essence, a program that is executing. Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a website. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details.

Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context

switching from one process to another is also costly. Threads, on the other hand, are lighter weight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost. While Java programs make use of process-based multitasking environments, process-based multitasking is not under Java's direct control. However, multithreaded multitasking is.

Multithreading enables you to write efficient programs that make maximum use of the processing power available in the system. One important way multithreading achieves this is by keeping idle time to a minimum. This is especially important for the interactive, networked environment in which Java operates because idle time is common. For example, the transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a much slower pace than they can be processed by the CPU. And, of course, user input is much slower than the computer. In a single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one—even though most of the time the program is idle, waiting for input. Multithreading helps you reduce this idle time because another thread can run when one is waiting.

If you have programmed for operating systems such as Windows, then you are already familiar with multithreaded programming. However, the fact that Java manages threads makes multithreading especially convenient because many of the details are handled for you.

The Java Thread Model

The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.

The value of a multithreaded environment is best understood in contrast to its counterpart. Single-threaded systems often use an approach called an *event loop* with *polling*. In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next. Once this polling mechanism returns with, say, a signal that a network file is

ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the program. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed. In general, in a single-threaded environment, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, the entire program stops running. This is an example of how only being able to do one thing at a time, something common to all single-threaded systems, whatever approach they use, is an inefficient way to use the CPU.

The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program. For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

As most readers know, over the past few years, multicore systems have become commonplace. Of course, single-core systems are still in use. It is important to understand that Java's multithreading features work in both types of systems. In a single-core system, concurrently executing threads share the CPU, with each thread receiving a slice of CPU time. Therefore, in a single-core system, two or more threads do not actually run at the same time, but idle CPU time is utilized. However, in multicore systems, it is possible for two or more threads to actually execute simultaneously. In many cases, this can further improve program efficiency and increase the speed of certain operations.

Java has straddled these two CPU architectures with the thread model by implementing Java threads, each as a wrapper around an OS thread. Such Java threads are known as platform threads. For most developers, platform threads, when used correctly, give great scalability and performance characteristics. However, for some high-throughput server applications, particularly ones managing many connections simultaneously like a heavily trafficked web application, this model can lead to performance bottlenecks. For these special kinds of applications, JDK 21 has added a new way to map Java threads onto OS threads. Java threads using this new feature are called *virtual threads*. When an application uses virtual threads, the JVM is

freed from tying each virtual thread one-to-one with an OS thread, offering greater efficiencies for highly multithreaded applications. While most developers will continue to use the default platform thread model, for those who do wish to explore virtual threads, fortunately, the programming model for the new virtual threads is the same as for platform threads. We'll cover virtual threads at the end of the chapter.

NOTE In addition to the multithreading features described in this chapter, you will also want to explore the Fork/Join Framework. It provides a powerful means of creating multithreaded applications that automatically scale to make best use of multicore environments. The Fork/Join Framework is part of Java's support for *parallel programming*, which is the name commonly given to the techniques that optimize some types of algorithms for parallel execution in systems that have more than one CPU. For a discussion of the Fork/Join Framework and other concurrency utilities, see [Chapter 29](#). Java's traditional multithreading capabilities are described here.

Threads exist in several states. Here is a general description: A thread can be *running*. It can be *ready to run* as soon as it gets CPU time. A running thread can be *suspended*, which temporarily halts its activity. A suspended thread can then be *resumed*, allowing it to pick up where it left off. A thread can be *blocked* when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch*. The rules that determine when a *context switch* takes place are simple:

- *A thread can voluntarily relinquish control.* This occurs when explicitly yielding, sleeping, or when blocked. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.

- *A thread can be preempted by a higher-priority thread.* In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing—by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking*.

In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For some operating systems, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

CAUTION Portability problems can arise from the differences in the way that operating systems context-switch threads of equal priority.

Synchronization

Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it. For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the *monitor*. The monitor is a control mechanism first defined by C.A.R. Hoare. You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

In Java, there is no class “Monitor”; instead, each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. This enables you to write very clear and concise multithreaded code, because synchronization support is built into the language.

Messaging

After you divide your program into separate threads, you need to define how they will communicate with each other. When programming with some other languages, you must depend on the operating system to establish communication between threads. This, of course, adds overhead. By contrast, Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have. Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

The Thread Class and the Runnable Interface

Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. **Thread** encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the **Thread** instance that spawned it. To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.

The **Thread** class defines several methods that help manage threads. Several of those used in this chapter are shown here:

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

Thus far, all the examples in this book have used a single thread of execution. The remainder of this chapter explains how to use **Thread** and

Runnable to create and manage threads, beginning with the one thread that all Java programs have: the main thread.

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other “child” threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread()**, which is a **public static** member of **Thread**. Its general form is shown here:

```
static Thread currentThread()
```

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Let's begin by reviewing the following example:

```
// Controlling the main Thread.
class CurrentThreadDemo {
    public static void main(String[] args) {
        Thread t = Thread.currentThread();

        System.out.println("Current thread: " + t);

        // change the name of the thread
        t.setName("My Thread");
        System.out.println("After name change: " + t);
```

```
try {
    for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
    }
} catch (InterruptedException e) {
    System.out.println("Main thread interrupted");
}
}
```

In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread()**, and this reference is stored in the local variable **t**. Next, the program displays information about the thread. The program then calls **setName()** to change the internal name of the thread. Information about the thread is then redisplayed. Next, a loop counts down from five, pausing one second between each line. The pause is accomplished by the **sleep()** method. The argument to **sleep()** specifies the delay period in milliseconds. Notice the **try/catch** block around this loop. The **sleep()** method in **Thread** might throw an **InterruptedException**. This would happen if some other thread wanted to interrupt this sleeping one. This example just prints a message if it gets interrupted. In a real program, you would need to handle this differently. Here is the output generated by this program:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Notice the output produced when `t` is used as an argument to `println()`. This displays, in order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is `main`. Its priority is 5, which is the default value, and `main` is also the name of the group of threads to which this thread belongs. A *thread group* is a data structure that controls the state of a collection of threads as a whole. After the name of the

thread is changed, **t** is again output. This time, the new name of the thread is displayed.

Let's look more closely at the methods defined by **Thread** that are used in the program. The **sleep()** method causes the thread from which it is called to suspend execution for the specified period of milliseconds. Its general form is shown here:

```
static void sleep(long milliseconds) throws InterruptedException
```

The number of milliseconds to suspend is specified in *milliseconds*. This method may throw an **InterruptedException**.

The **sleep()** method has a second form, shown next, that allows you to specify the period in terms of milliseconds and nanoseconds:

```
static void sleep(long milliseconds, int nanoseconds) throws  
InterruptedException
```

This second form is useful only in environments that allow timing periods as short as nanoseconds.

As the preceding program shows, you can set the name of a thread by using **setName()**. You can obtain the name of a thread by calling **getName()** (but note that this is not shown in the program). These methods are members of the **Thread** class and are declared like this:

```
final void setName(String threadName)  
final String getName()
```

Here, *threadName* specifies the name of the thread.

Creating a Thread

In the most general sense, you create a thread by instantiating an object of type **Thread**. Java defines two ways in which this can be accomplished:

- You can implement the **Runnable** interface.
- You can extend the **Thread** class itself.

The following two sections look at each method, in turn.

Implementing Runnable

The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called **run()**, which is declared like this:

```
public void run( )
```

Inside **run()**, you will define the code that constitutes the new thread. It is important to understand that **run()** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run()** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run()** returns.

After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we will use is shown here:

```
Thread(Runnable threadOb, String threadName)
```

In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*.

After the new thread is created, it will not start running until you call its **start()** method, which is declared within **Thread**. In essence, **start()** initiates a call to **run()**. The **start()** method is shown here:

```
void start( )
```

Here is an example that creates a new thread and starts it running:

```
// Create a second thread.
class NewThread implements Runnable {
    Thread t;
```

```
NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
}

// This is the entry point for the second thread.
public void run() {
    try {
        for(int i = 5; i > 0; i--) {
            System.out.println("Child Thread: " + i);
            Thread.sleep(500);
        }
    } catch (InterruptedException e) {
        System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
}

}

class ThreadDemo {
    public static void main(String[] args) {
        NewThread nt = new NewThread(); // create a new thread
        nt.t.start(); // Start the thread

        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

Inside **NewThread**'s constructor, a new **Thread** object is created by the following statement:

```
t = new Thread(this, "Demo Thread");
```

Passing **this** as the first argument indicates that you want the new thread to call the **run()** method on the **this** object. Inside **main()**, **start()** is called, which starts the thread of execution beginning at the **run()** method. This causes the child thread's **for** loop to begin. Next, the main thread enters its **for** loop. Both threads continue running, sharing the CPU in single-core systems, until their loops finish. The output produced by this program is as follows.(Your output may vary based upon the specific execution environment.)

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4

Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

As mentioned earlier, in a multithreaded program, it is often useful for the main thread to be the last thread to finish running. The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to terminate earlier than the main thread. Shortly, you will see a better way to wait for a thread to finish.

Extending Thread

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class. The extending class must override the **run()** method, which is the entry point for the new

thread. As before, a call to **start()** begins execution of the new thread. Here is the preceding program rewritten to extend **Thread**:

```
// Create a second thread by extending Thread
class NewThread extends Thread {

    NewThread() {
        // Create a new, second thread
        super("Demo Thread");
        System.out.println("Child thread: " + this);
    }

    // This is the entry point for the second thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

class ExtendThread {
    public static void main(String[] args) {
        NewThread nt = new NewThread(); // create a new thread
        nt.start(); // start the thread
    }
}
```

```

try {
    for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
    }
} catch (InterruptedException e) {
    System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}

```

This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**.

Notice the call to **super()** inside **NewThread**. This invokes the following form of the **Thread** constructor:

```
public Thread(String threadName)
```

Here, *threadName* specifies the name of the thread.

Choosing an Approach

At this point, you might be wondering why Java has two ways to create child threads, and which approach is better. The answers to these questions turn on the same point. The **Thread** class defines several methods that can be overridden by a derived class. Of these methods, the only one that *must* be overridden is **run()**. This is, of course, the same method required when you implement **Runnable**. Many Java programmers feel that classes should be extended only when they are being enhanced or adapted in some way. So, if you will not be overriding any of **Thread**'s other methods, it is probably best simply to implement **Runnable**. Also, by implementing **Runnable**, your thread class does not need to inherit **Thread**, making it free to inherit a different class. Ultimately, which approach to use is up to you. However, throughout the rest of this chapter, we will create threads by using classes that implement **Runnable**.

Creating Multiple Threads

So far, you have been using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs. For example, the following program creates three child threads:

```
// Create multiple threads.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
    }
}
```

```
// This is the entry point for thread.
public void run() {
    try {
        for(int i = 5; i > 0; i--) {
            System.out.println(name + ": " + i);
            Thread.sleep(1000);
        }
    } catch (InterruptedException e) {
        System.out.println(name + "Interrupted");
    }
    System.out.println(name + " exiting.");
}
}

class MultiThreadDemo {
    public static void main(String[] args) {
        NewThread nt1 = new NewThread("One");
        NewThread nt2 = new NewThread("Two");
        NewThread nt3 = new NewThread("Three");

        // Start the threads.
        nt1.t.start();
        nt2.t.start();
        nt3.t.start();

        try {
            // wait for other threads to end
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        System.out.println("Main thread exiting.");
    }
}
```

Sample output from this program is shown here. (Your output may vary based upon the specific execution environment.)

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2

Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

As you can see, once started, all three child threads share the CPU. Notice the call to **sleep(10000)** in **main()**. This causes the main thread to sleep for ten seconds and ensures that it will finish last.

Using **isAlive()** and **join()**

As mentioned, often you will want the main thread to finish last. In the preceding examples, this is accomplished by calling **sleep()** within **main()**, with a long enough delay to ensure that all child threads terminate prior to the main thread. However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended? Fortunately, **Thread** provides a means by which you can answer this question.

Two ways exist to determine whether a thread has finished. First, you can call **isAlive()** on the thread. This method is defined by **Thread**, and its general form is shown here:

```
final boolean isAlive()
```

The **isAlive()** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.

While **isAlive()** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join()**, shown here:

```
final void join() throws InterruptedException
```

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it. Additional forms of **join()** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Here is an improved version of the preceding example that uses **join()** to ensure that the main thread is the last to finish. It also demonstrates the **isAlive()** method.

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
    }
}
```

```
// This is the entry point for thread.
public void run() {
    try {
        for(int i = 5; i > 0; i--) {
            System.out.println(name + ": " + i);
            Thread.sleep(1000);
        }
    } catch (InterruptedException e) {
        System.out.println(name + " interrupted.");
    }
    System.out.println(name + " exiting.");
}
}

class DemoJoin {
    public static void main(String[] args) {
        NewThread nt1 = new NewThread("One");
        NewThread nt2 = new NewThread("Two");
        NewThread nt3 = new NewThread("Three");

        // Start the threads.
        nt1.t.start();
        nt2.t.start();
        nt3.t.start();

        System.out.println("Thread One is alive: "
                           + nt1.t.isAlive());
        System.out.println("Thread Two is alive: "
                           + nt2.t.isAlive());
        System.out.println("Thread Three is alive: "
                           + nt3.t.isAlive());
        // wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish.");
            nt1.t.join();
            nt2.t.join();
            nt3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
    }
}
```

Sample output from this program is shown here. (Your output may vary based upon the specific execution environment.)

```
+ nt1.t.isAlive();
+ nt2.t.isAlive());
New thread: Thread[One,5,main]  e is alive: "
New thread: Thread[Two,5,main]  live());
New thread: Thread[Three,5,main]
Thread One is alive: true      exiting.");
Thread Two is alive: true
Thread Three is alive: true
} Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

As you can see, after the calls to **join()** return, the threads have stopped executing.

Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, over a given period of time,

higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

In theory, threads of equal priority should get equal access to the CPU. But you need to be careful. Remember, Java is designed to work in a wide range of environments. Some of those environments implement multitasking fundamentally differently than others. You should think of the thread priority as a polite suggestion to the thread scheduler as to how to prioritize your threads at times when the system resources are getting stretched. But the thread scheduler will likely have other factors to consider as well. So you should experiment with your application in a range of conditions if you are using thread priorities and not depend on it as if it were a strict rule.

To set a thread's priority, use the `setPriority()` method, which is a member of **Thread**. This is its general form:

```
final void setPriority(int level)
```

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5. These priorities are defined as **static final** variables within **Thread**.

You can obtain the current priority setting by calling the `getPriority()` method of **Thread**, shown here:

```
final int getPriority()
```

Implementations of Java may have radically different behavior when it comes to scheduling. Most of the inconsistencies arise when you have threads that are relying on preemptive behavior, instead of cooperatively

giving up CPU time. The safest way to obtain predictable, cross-platform behavior with Java is to use threads that voluntarily give up control of the CPU.

Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization*. As you will see, Java provides unique, language-level support for it.

Key to synchronization is the concept of the monitor. A *monitor* is an object that is used as a mutually exclusive lock. Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

You can synchronize your code in either of two ways. Both involve the use of the **synchronized** keyword, and both are examined here.

Using Synchronized Methods

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

To understand the need for synchronization, let's begin with a simple example that does not use it—but should. The following program has three simple classes. The first one, **Callme**, has a single method named **call()**. The **call()** method takes a **String** parameter called **msg**. This method tries to print the **msg** string inside of square brackets. The interesting thing to notice is that after **call()** prints the opening bracket and the **msg** string, it calls **Thread.sleep(1000)**, which pauses the current thread for one second.

The constructor of the next class, **Caller**, takes a reference to an instance of the **Callme** class and a **String**, which are stored in **target** and **msg**, respectively. The constructor also creates a new thread that will call this object's **run()** method. The **run()** method of **Caller** calls the **call()** method on the **target** instance of **Callme**, passing in the **msg** string. Finally, the **Synch** class starts by creating a single instance of **Callme**, and three instances of **Caller**, each with a unique message string. The same instance of **Callme** is passed to each **Caller**.

```
// This program is not synchronized.
class Callme {
    void call(String msg) {
        System.out.print("[ " + msg);
        try {
            Thread.sleep(1000);
        } catch(InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("] ");
    }
}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;

    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
    }

    public void run() {
        target.call(msg);
    }
}

class Synch {
    public static void main(String[] args) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");
    }
}
```

```
// Start the threads.  
ob1.t.start();  
ob2.t.start();  
ob3.t.start();  
  
// wait for threads to end  
try {  
    ob1.t.join();  
    ob2.t.join();  
    ob3.t.join();  
} catch(InterruptedException e) {  
    System.out.println("Interrupted");  
}  
}  
}  
}
```

Here is the output produced by this program:

```
[Hello [Synchronized [World]  
]  
]  
]
```

As you can see, by calling **sleep()**, the **call()** method allows execution to switch to another thread. This results in the mixed-up output of the three message strings. In this program, nothing exists to prevent all three threads from calling the same method, on the same object, at the same time. This is known as a *race condition*, because the three threads are racing each other to complete the method. This example used **sleep()** to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will occur. This can cause a program to run right one time and wrong the next.

To fix the preceding program, you must *serialize* access to **call()**. That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede **call()**'s definition with the keyword **synchronized**, as shown here:

```
class Callme {  
    synchronized void call(String msg) {  
        ...  
    }  
}
```

This prevents other threads from entering `call()` while another thread is using it. After `synchronized` has been added to `call()`, the output of the program is as follows:

```
[Hello]  
[Synchronized]  
[World]
```

Any time that you have a method, or group of methods, that changes the internal state of an object in a multithreaded situation, you should consider using the `synchronized` keyword to guard the state from race conditions. Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, nonsynchronized methods on that instance will continue to be callable.

The `synchronized` Statement

While creating `synchronized` methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following: Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use `synchronized` methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add `synchronized` to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a `synchronized` block.

This is the general form of the `synchronized` statement:

```
synchronized(objRef) {  
    // statements to be synchronized  
}
```

Here, *objRef* is a reference to the object being synchronized. A synchronized block ensures that a call to a synchronized method that is a

member of *objRef*'s *class* occurs only after the current thread has successfully entered *objRef*'s monitor.

Here is an alternative version of the preceding example, using a synchronized block within the **run()** method:

```
// This program uses a synchronized block.
class Callme {
    void call(String msg) {
        System.out.print("[ " + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("] ");
    }
}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;

    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
    }

    // synchronize calls to call()
    public void run() {
        synchronized(target) { // synchronized block
            target.call(msg);
        }
    }
}
```

```

        }
    }
}

class Synch1 {
    public static void main(String[] args) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");

        // Start the threads.
        ob1.t.start();
        ob2.t.start();
        ob3.t.start();

        // wait for threads to end
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch(InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
}

```

Here, the `call()` method is not modified by `synchronized`. Instead, the `synchronized` statement is used inside `Caller's run()` method. This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

Interthread Communication

The preceding examples unconditionally blocked other threads from asynchronous access to certain methods. This use of the implicit monitors in Java objects is powerful, but you can achieve a more subtle level of control through interthread communication. As you will see, this is especially easy in Java.

As discussed earlier, multithreading replaces event loop programming by dividing your tasks into discrete, logical units. Threads also provide a secondary benefit: they do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time. For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.

To avoid polling, Java includes an elegant interthread communication mechanism via the **wait()**, **notify()**, and **notifyAll()** methods. These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context. Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:

- **wait()** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify()** or **notifyAll()**.
- **notify()** wakes up a thread that called **wait()** on the same object.
- **notifyAll()** wakes up all the threads that called **wait()** on the same object. One of the threads will be granted access.

These methods are declared within **Object**, as shown here:

```
final void wait() throws InterruptedException  
final void notify()  
final void notifyAll()
```

Additional forms of **wait()** exist that allow you to specify a period of time to wait.

Before working through an example that illustrates interthread communication, an important point needs to be made. Although **wait()**

normally waits until **notify()** or **notifyAll()** is called, there is a possibility that in very rare cases the waiting thread could be awakened due to a *spurious wakeup*. In this case, a waiting thread resumes without **notify()** or **notifyAll()** having been called. (In essence, the thread resumes for no apparent reason.) Because of this remote possibility, the Java API documentation recommends that calls to **wait()** should take place within a loop that checks the condition on which the thread is waiting. The following example shows this technique.

Let's now work through an example that uses **wait()** and **notify()**. To begin, consider the following sample program that incorrectly implements a simple form of the producer/consumer problem. It consists of four classes: **Q**, the queue that you're trying to synchronize; **Producer**, the threaded object that is producing queue entries; **Consumer**, the threaded object that is consuming queue entries; and **PC**, the tiny class that creates the single **Q**, **Producer**, and **Consumer**.

```
// An incorrect implementation of a producer and consumer.
class Q {
    int n;

    synchronized int get() {
        System.out.println("Got: " + n);
        return n;
    }

    synchronized void put(int n) {
        this.n = n;
        System.out.println("Put: " + n);
    }
}

class Producer implements Runnable {
    Q q;
    Thread t;
```

```
Producer(Q q) {
    this.q = q;
    t = new Thread(this, "Producer");
}

public void run() {
    int i = 0;

    while(true) {
        q.put(i++);
    }
}
}

class Consumer implements Runnable {
    Q q;
    Thread t;

    Consumer(Q q) {
        this.q = q;
        t = new Thread(this, "Consumer");
    }

    public void run() {
        while(true) {
            q.get();
        }
    }
}

class PC {
    public static void main(String[] args) {
        Q q = new Q();
        Producer p = new Producer(q);
        Consumer c = new Consumer(q);

        // Start the threads.
        p.t.start();
        c.t.start();

        System.out.println("Press Control-C to stop.");
    }
}
```

Although the **put()** and **get()** methods on **Q** are synchronized, nothing stops the producer from overrunning the consumer, nor will anything prevent the consumer from consuming the same queue value twice. Thus, you get the erroneous output shown here (the exact output will vary with processor speed and task load):

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

As you can see, after the producer put 1, the consumer started and got the same 1 five times in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer have a chance to consume them.

The proper way to write this program in Java is to use **wait()** and **notify()** to signal in both directions, as shown here:

```
// A correct implementation of a producer and consumer.
class Q {
    int n;
    boolean valueSet = false;

    synchronized int get() {
        while(!valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }

        System.out.println("Got: " + n);
        valueSet = false;
        notify();
        return n;
    }

    synchronized void put(int n) {
        while(valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }

        this.n = n;
        valueSet = true;
        System.out.println("Put: " + n);
        notify();
    }
}

class Producer implements Runnable {
    Q q;
    Thread t;
```

```
Producer(Q q) {
    this.q = q;
    t = new Thread(this, "Producer");
}

public void run() {
    int i = 0;

    while(true) {
        q.put(i++);
    }
}
}

class Consumer implements Runnable {
    Q q;
    Thread t;

    Consumer(Q q) {
        this.q = q;
        t = new Thread(this, "Consumer");
    }

    public void run() {
        while(true) {
            q.get();
        }
    }
}

class PCFixed {
    public static void main(String[] args) {
        Q q = new Q();
        Producer p = new Producer(q);
        Consumer c = new Consumer(q);

        // Start the threads.
        p.t.start();
        c.t.start();

        System.out.println("Press Control-C to stop.");
    }
}
```

Inside `get()`, `wait()` is called. This causes its execution to suspend until **Producer** notifies you that some data is ready. When this happens, execution inside `get()` resumes. After the data has been obtained, `get()` calls `notify()`. This tells **Producer** that it is okay to put more data in the queue. Inside `put()`, `wait()` suspends execution until **Consumer** has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and `notify()` is called. This tells **Consumer** that it should now remove it.

Here is some output from this program, which shows the clean synchronous behavior:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

Deadlock

A special type of error that you need to avoid that relates specifically to multitasking is *deadlock*, which occurs when two threads have a circular dependency on a pair of synchronized objects. For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete. Deadlock is a difficult error to debug for two reasons:

- In general, it occurs unpredictably, when the two threads time-slice in just the right way.
- It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of

events than just described.)

To understand deadlock fully, it is useful to see it in action. The next example creates two classes, **A** and **B**, with methods **foo()** and **bar()**, respectively, which pause briefly before trying to call a method in the other class. The main class, named **Deadlock**, creates an **A** and a **B** instance, and then calls **deadlockStart()** to start a second thread that sets up the deadlock condition. The **foo()** and **bar()** methods use **sleep()** as a way to force the deadlock condition to occur.

```
// An example of deadlock.
class A {
    synchronized void foo(B b) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered A.foo");
        try {
            Thread.sleep(1000);
        } catch(Exception e) {
            System.out.println("A Interrupted");
        }
        System.out.println(name + " trying to call B.last()");
        b.last();
    }
}
```

```
synchronized void last() {
    System.out.println("Inside A.last");
}
}

class B {
    synchronized void bar(A a) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered B.bar");

        try {
            Thread.sleep(1000);
        } catch(Exception e) {
            System.out.println("B Interrupted");
        }

        System.out.println(name + " trying to call A.last()");
        a.last();
    }

    synchronized void last() {
        System.out.println("Inside B.last");
    }
}

class Deadlock implements Runnable {
    A a = new A();
    B b = new B();
    Thread t;

    Deadlock() {
        Thread.currentThread().setName("MainThread");
        t = new Thread(this, "RacingThread");
    }

    void deadlockStart() {
        t.start();
        a.foo(b); // get lock on a in this thread.
        System.out.println("Back in main thread");
    }
}
```

When you run this program, you will see the output shown here, although whether **A.foo()** or **B.bar()** executes first will vary based on the specific execution environment.

```
MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call B.last()
RacingThread trying to call A.last()
```

Because the program has deadlocked, you need to press CTRL-C to end the program. You can see a full thread and monitor cache dump by pressing CTRL-BREAK on a PC. You will see that **RacingThread** owns the monitor on **b**, while it is waiting for the monitor on **a**. At the same time, **MainThread** owns **a** and is waiting to get **b**. This program will never complete. As this example illustrates, if your multithreaded program locks up occasionally, deadlock is one of the first conditions that you should check for.

Suspending, Resuming, and Stopping Threads

Sometimes, suspending execution of a thread is useful. For example, a separate thread can be used to display the time of day. If the user doesn't want a clock, then its thread can be suspended. Whatever the case, suspending a thread is a simple matter. Once it's suspended, restarting the thread is also a simple matter.

The mechanisms to suspend, stop, and resume threads differ between early versions of Java, such as Java 1.0, and more modern versions, beginning with Java 2. Prior to Java 2, a program used **suspend()**, **resume()**, and **stop()**, which are methods defined by **Thread**, to pause, restart, and stop the execution of a thread, respectively. Although these methods seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs. Here's why. The **suspend()** method of the **Thread** class was deprecated by Java 2 several years ago, and will be removed in a future release. This was done because **suspend()** can sometimes cause serious system failures. Assume that a thread has obtained locks on critical data structures. If that thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked.

The **resume()** method is also deprecated. It does not cause problems, but it cannot be used without the **suspend()** method as its counterpart.

The **stop()** method of the **Thread** class, too, was deprecated by Java 2, and will be removed in a future release. This was done because this method can sometimes cause serious system failures. Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state. The trouble is that **stop()** causes any lock the calling thread holds to be released. Thus, the corrupted data might be used by another thread that is waiting on the same lock.

Because you can't now use the **suspend()**, **resume()**, or **stop()** methods to control a thread, you might be thinking that no way exists to pause, restart, or terminate a thread. But, fortunately, this is not true. Instead, a thread must be designed so that the **run()** method periodically checks to determine whether that thread should suspend, resume, or stop its own execution. Typically, this is accomplished by establishing a flag variable that indicates the execution state of the thread. As long as this flag is set to "running," the **run()** method must continue to let the thread execute. If this variable is set to "suspend," the thread must pause. If it is set to "stop," the thread must terminate. Of course, a variety of ways exist in which to write such code, but the central theme will be the same for all programs.

The following example illustrates how the **wait()** and **notify()** methods that are inherited from **Object** can be used to control the execution of a thread. Let us consider its operation. The **NewThread** class contains a **boolean** instance variable named **suspendFlag**, which is used to control the execution of the thread. It is initialized to **false** by the constructor. The **run()** method contains a **synchronized** statement block that checks **suspendFlag**. If that variable is **true**, the **wait()** method is invoked to suspend the execution of the thread. The **mysuspend()** method sets **suspendFlag** to **true**. The **myresume()** method sets **suspendFlag** to **false** and invokes **notify()** to wake up the thread. Finally, the **main()** method has been modified to invoke the **mysuspend()** and **myresume()** methods.

```
// Suspending and resuming a thread the modern way.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    boolean suspendFlag;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        suspendFlag = false;
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 15; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(200);
                synchronized(this) {
                    while(suspendFlag) {
                        wait();
                    }
                }
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }

    synchronized void mysuspend() {
        suspendFlag = true;
    }

    synchronized void myresume() {
        suspendFlag = false;
        notify();
    }
}
```

```
class SuspendResume {
    public static void main(String[] args) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");

        ob1.t.start(); // Start the thread
        ob2.t.start(); // Start the thread

        try {
            Thread.sleep(1000);
            ob1.mysuspend();
            System.out.println("Suspending thread One");
            Thread.sleep(1000);
            ob1.myresume();
            System.out.println("Resuming thread One");
            ob2.mysuspend();
            System.out.println("Suspending thread Two");
            Thread.sleep(1000);
            ob2.myresume();
            System.out.println("Resuming thread Two");
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        // wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish.");
            ob1.t.join();
            ob2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        System.out.println("Main thread exiting.");
    }
}
```

When you run the program, you will see the threads suspend and resume. Later in this book, you will see more examples that use the modern mechanism of thread control. Although this mechanism may not appear as

simple to use as the old way, nevertheless, it is the way required to ensure that run-time errors don't occur. It is the approach that *must* be used for all new code.

Obtaining a Thread's State

As mentioned earlier in this chapter, a thread can exist in a number of different states. You can obtain the current state of a thread by calling the `getState()` method defined by **Thread**. It is shown here:

```
Thread.State getState()
```

It returns a value of type **Thread.State** that indicates the state of the thread at the time at which the call was made. **State** is an enumeration defined by **Thread**. (An enumeration is a list of named constants. It is discussed in detail in [Chapter 12](#).) Here are the values that can be returned by `getState()`:

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called <code>sleep()</code> . This state is also entered when a timeout version of <code>wait()</code> or <code>join()</code> is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of <code>wait()</code> or <code>join()</code> .

[Figure 11-1](#) diagrams how the various thread states relate.

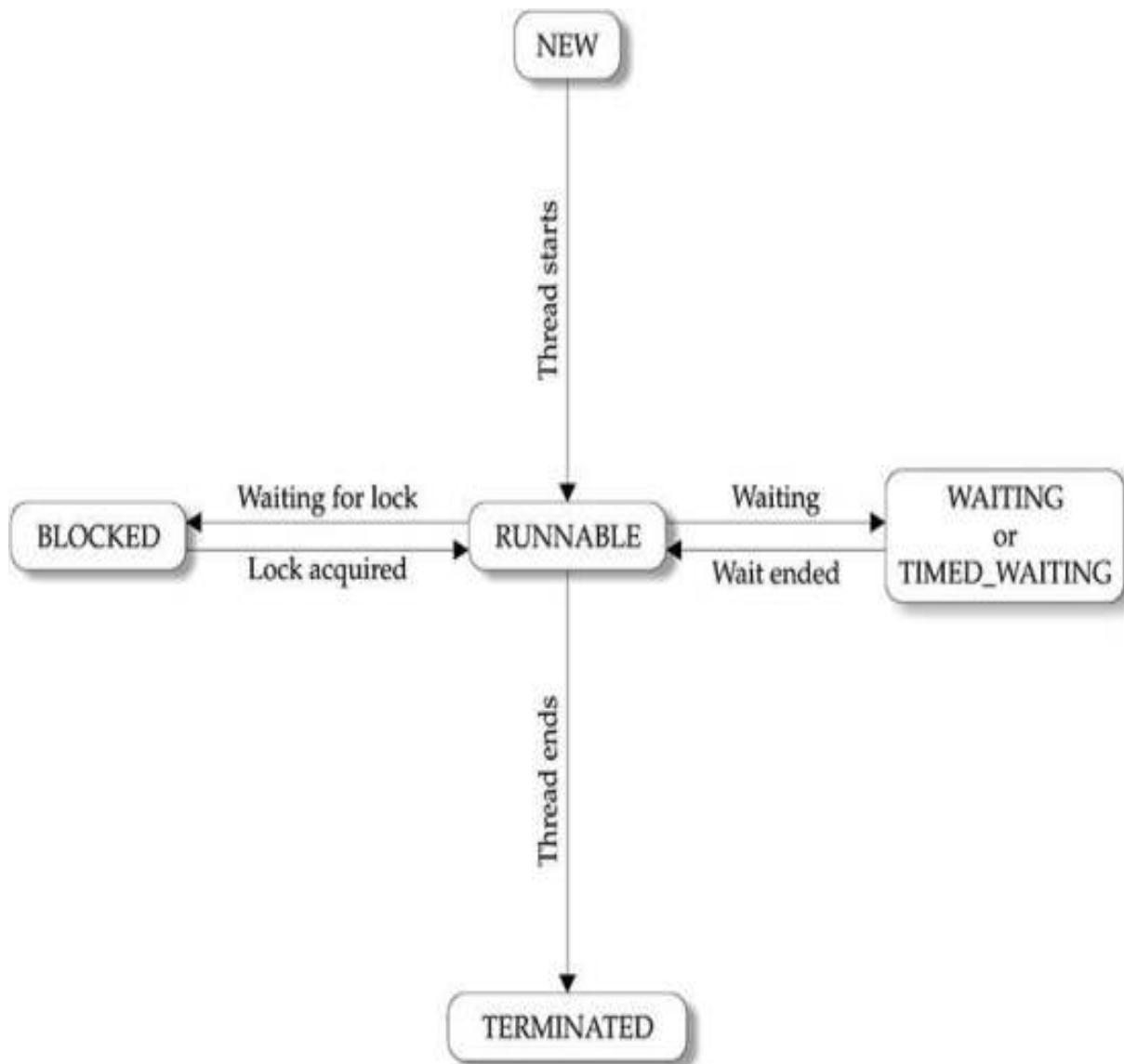


Figure 11-1 Thread states

Given a **Thread** instance, you can use **getState()** to obtain the state of a thread. For example, the following sequence determines if a thread called **thrd** is in the **RUNNABLE** state at the time **getState()** is called:

```

Thread.State ts = thrd.getState();

if(ts == Thread.State.RUNNABLE) // ...
  
```

It is important to understand that a thread's state may change after the call to **getState()**. Thus, depending on the circumstances, the state obtained by

calling `getState()` may not reflect the actual state of the thread only a moment later. For this (and other) reasons, `getState()` is not intended to provide a means of synchronizing threads. It's primarily used for debugging or for profiling a thread's run-time characteristics.

Using a Factory Method to Create and Start a Thread

In some cases, it is not necessary to separate the creation of a thread from the start of its execution. In other words, sometimes it is convenient to create and start a thread at the same time. One way to do this is to use a static factory method. A *factory method* is a method that returns an object of a class. Typically, factory methods are *static* methods of a class. They are used for a variety of reasons, such as to set an object to some initial state prior to use, to configure a specific type of object, or in some cases to enable an object to be reused. As it relates to creating and starting a thread, a factory method will create the thread, call `start()` on the thread, and then return a reference to the thread. With this approach, you can create and start a thread through a single method call, thus streamlining your code.

For example, assuming the **ThreadDemo** program shown near the start of this chapter, adding the following factory method to **NewThread** enables you to create and start a thread in a single step:

```
// A factory method that creates and starts a thread.
public static NewThread createAndStart() {
    NewThread myThrd = new NewThread();
    myThrd.t.start();
    return myThrd;
}
```

Using `createAndStart()`, you can now replace the sequence

```
NewThread nt = new NewThread(); // create a new thread
nt.t.start(); // Start the thread
```

with

```
NewThread nt = NewThread.createAndStart();
```

Now the thread is created and started in one step.

In cases in which you don't need to keep a reference to the executing thread, you can sometimes create and start a thread with one line of code, without the use of a factory method. For example, again assuming the **ThreadDemo** program, the following creates and starts a **NewThread** thread:

```
new NewThread().t.start();
```

However, in real-world applications, you will usually need to keep a reference to the thread, so the factory method is often a good choice.

Virtual Threads

As mentioned at the beginning of this chapter, the Java platform implements Java threads as wrappers around the underlying OS's native threads. In this model, when the OS runs out of threads, so does Java! For high-throughput, high-scale applications such as application servers, which generally like to use a new thread for each new incoming request, this can be a performance bottleneck as to the number of requests the server can handle at the same time. JDK 21 has introduced a new way to map Java threads onto OS threads in which many Java threads can share the same OS thread. This is called *virtual threading*. By scheduling work on the OS in a more flexible way, high-scale server applications can perform better because they can handle more concurrent requests gracefully. The number of virtual Java threads an application can create is no longer limited to the number of OS threads available to Java.

Fortunately, the programming model for using platform threads and virtual threads is the same except for how the Java threads are first created and managed when the JVM terminates. All the examples so far have created threads that are platform threads, since these are the kind of threads that work well for most applications. You can determine what kind of thread is in use by the **Thread** class method

boolean isVirtual()

which is **false** for platform threads and **true** for virtual threads.

In order to create a virtual thread in Java, you can use the static **Thread** method **ofVirtual()**, from which you can obtain a **Thread.Builder** object from which to create virtual threads. For example, the code

```
Runnable myTask = // a task within the run() method  
Thread myVirtualThread = Thread.ofVirtual().start(myTask);
```

shows a **Runnable** task being run on a virtual thread. The virtual thread is an instance of the Java **Thread** class you already learned about in this chapter, and so it has the same life cycle and API methods for managing execution. It is important to note that if you want to use virtual threads, you have to explicitly choose this API to do so.

The companion to **ofVirtual()** is the static **Thread** method **ofPlatform()**, which creates a **Thread.Builder** object from which you can create traditional platform threads. However, since platform threads are the default type of threads, the APIs that you have already seen create platform threads, such as

```
Runnable myTask = // a task within the run() method  
Thread myThread = new Thread(task, "myTask");
```

or the default no argument constructor of the **Thread** class.

While virtual threads “look like” platform threads for the most part, they differ in that they are not named by default (their name variable is the empty string) and the JVM does not wait for them to finish their work when it exits.

If a virtual thread is waiting, the Java platform can schedule its OS thread to be useful for another virtual thread that needs attention. In this way, the Java platform can use OS threads more efficiently for applications that have tasks spending significant time waiting on network connections or blocking I/O. For applications mostly doing continuous processing, like large algorithmic calculations, virtual threads will likely not bring much benefit. So virtual threads, while easy to use because they “look like” traditional threads, bring performance benefits only for certain types of applications and will likely require judicious use together with performance testing to realize their full potential.

Using Multithreading

The key to utilizing Java's multithreading features effectively is to think concurrently rather than serially. For example, when you have two subsystems within a program that can execute concurrently, make them individual threads. With the careful use of multithreading, you can create very efficient programs. A word of caution is in order, however: If you create too many threads, you can actually degrade the performance of your program rather than enhance it. Remember, some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program! One last point: To create compute-intensive applications that can automatically scale to make use of the available processors in a multicore system, consider using the Fork/Join Framework, which is described in [Chapter 29](#).

CHAPTER

12

Enumerations, Autoboxing, and Annotations

This chapter examines three features that were not originally part of Java, but over time each has become a near indispensable aspect of Java programming: enumerations, autoboxing, and annotations. Originally added by JDK 5, each is a feature upon which Java programmers have come to rely because each offers a streamlined approach to handling common programming tasks. This chapter also discusses Java's type wrappers and introduces reflection.

Enumerations

In its simplest form, an *enumeration* is a list of named constants that define a new data type and its legal values. Thus, an enumeration object can hold only a value that was declared in the list. Other values are not allowed. In other words, an enumeration gives you a way to explicitly specify the only values that a data type can legally have. Enumerations are commonly used to define a set of values that represent a collection of items. For example, you might use an enumeration to represent the error codes that can result from some operation, such as success, failed, or pending; or a list of the states that a device might be in, such as running, stopped, or paused. In early versions of Java, such values were defined using **final** variables, but enumerations offer a far superior approach.

Although Java enumerations might, at first glance, appear similar to enumerations in other languages, this similarity may be only skin deep

because, in Java, an enumeration defines a class type. By making enumerations into classes, the capabilities of the enumeration are greatly expanded. For example, in Java, an enumeration can have constructors, methods, and instance variables. Because of their power and flexibility, enumerations are widely used throughout the Java API library.

Enumeration Fundamentals

An enumeration is created using the **enum** keyword. For example, here is a simple enumeration that lists various apple varieties:

```
// An enumeration of apple varieties.  
enum Apple {  
    Jonathan, GoldenDel, RedDel, Winesap, Cortland  
}
```

The identifiers **Jonathan**, **GoldenDel**, and so on are called *enumeration constants*. Each is implicitly declared as a public, static final member of **Apple**. Furthermore, their type is the type of the enumeration in which they are declared, which is **Apple** in this case. Thus, in the language of Java, these constants are called *self-typed*, in which “self” refers to the enclosing enumeration.

Once you have defined an enumeration, you can create a variable of that type. However, even though enumerations define a class type, you do not instantiate an **enum** using **new**. Instead, you declare and use an enumeration variable in much the same way as you do one of the primitive types. For example, this declares **ap** as a variable of enumeration type **Apple**:

```
Apple ap;
```

Because **ap** is of type **Apple**, the only values that it can be assigned (or can contain) are those defined by the enumeration. For example, this assigns **ap** the value **RedDel**:

```
ap = Apple.RedDel;
```

Notice that the symbol **RedDel** is preceded by **Apple**.

Two enumeration constants can be compared for equality by using the `=` relational operator. For example, this statement compares the value in `ap` with the **GoldenDel** constant:

```
if (ap == Apple.GoldenDel) // ...
```

An enumeration value can also be used to control a **switch** statement. Of course, all of the **case** statements must use constants from the same **enum** as that used by the **switch** expression. For example, this **switch** is perfectly valid:

```
// Use an enum to control a switch statement.
switch(ap) {
    case Jonathan:
        // ...
    case Winesap:
        // ...
```

Notice that in the **case** statements, the names of the enumeration constants are used without being qualified by their enumeration type name. That is, **Winesap**, not **Apple.Winesap**, is used. This is because the type of the enumeration in the **switch** expression has already implicitly specified the **enum** type of the **case** constants. There is no need to qualify the constants in the **case** statements with their **enum** type name. In fact, attempting to do so will cause a compilation error.

When an enumeration constant is displayed, such as in a **println()** statement, its name is output. For example, given the statement

```
System.out.println(Apple.Winesap);
```

the name **Winesap** is displayed.

The following program puts together all of the pieces and demonstrates the **Apple** enumeration:

```
// An enumeration of apple varieties.  
enum Apple {  
    Jonathan, GoldenDel, RedDel, Winesap, Cortland  
}  
  
class EnumDemo {  
    public static void main(String[] args)  
    {  
        Apple ap;  
  
        ap = Apple.RedDel;  
  
        // Output an enum value.  
        System.out.println("Value of ap: " + ap);  
        System.out.println();  
  
        ap = Apple.GoldenDel;  
  
        // Compare two enum values.  
        if(ap == Apple.GoldenDel)  
            System.out.println("ap contains GoldenDel.\n");  
  
        // Use an enum to control a switch statement.  
        switch(ap) {  
            case Jonathan:  
                System.out.println("Jonathan is red.");  
                break;  
            case GoldenDel:  
                System.out.println("Golden Delicious is yellow.");  
                break;  
            case RedDel:  
                System.out.println("Red Delicious is red.");  
                break;  
            case Winesap:  
                System.out.println("Winesap is red.");  
                break;  
            case Cortland:  
                System.out.println("Cortland is red.");  
                break;  
        }  
    }  
}
```

- The output from the program is shown here:

```
Value of ap: RedDel  
ap contains GoldenDel.  
Golden Delicious is yellow.
```

The values() and valueOf() Methods

All enumerations automatically contain two predefined methods: **values()** and **valueOf()**. Their general forms are shown here:

```
public static enum-type [ ] values()  
public static enum-type valueOf(String str)
```

The **values()** method returns an array that contains a list of the enumeration constants. The **valueOf()** method returns the enumeration constant whose value corresponds to the string passed in *str*. In both cases, *enum-type* is the type of the enumeration. For example, in the case of the **Apple** enumeration shown earlier, the return type of **Apple.valueOf("Winesap")** is **Winesap**.

The following program demonstrates the **values()** and **valueOf()** methods:

```
// Use the built-in enumeration methods.

// An enumeration of apple varieties.
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo2 {
    public static void main(String[] args)
    {
        Apple ap;

        System.out.println("Here are all Apple constants:");

        // use values()
        Apple[] allapples = Apple.values();
        for(Apple a : allapples)
            System.out.println(a);

        System.out.println();

        // use valueOf()
        ap = Apple.valueOf("Winesap");
        System.out.println("ap contains " + ap);
    }
}
```

The output from the program is shown here:

```
Here are all Apple constants:
Jonathan
GoldenDel
RedDel
Winesap
Cortland

ap contains Winesap
```

Notice that this program uses a for-each style **for** loop to cycle through the array of constants obtained by calling **values()**. For the sake of

illustration, the variable **allapples** was created and assigned a reference to the enumeration array. However, this step is not necessary because the **for** could have been written as shown here, eliminating the need for the **allapples** variable:

```
for(Apple a : Apple.values())
    System.out.println(a);
```

Now, notice how the value corresponding to the name **Winesap** was obtained by calling **valueOf()**:

```
ap = Apple.valueOf("Winesap");
```

As explained, **valueOf()** returns the enumeration value associated with the name of the constant represented as a string.

Java Enumerations Are Class Types

As mentioned, a Java enumeration is a class type. Although you don't instantiate an **enum** using **new**, it otherwise has much the same capabilities as other classes. The fact that **enum** defines a class gives the Java enumeration extraordinary power. For example, you can give it constructors, add instance variables and methods, and even implement interfaces.

It is important to understand that each enumeration constant is an object of its enumeration type. Thus, when you define a constructor for an **enum**, the constructor is called when each enumeration constant is created. Also, each enumeration constant has its own copy of any instance variables defined by the enumeration. For example, consider the following version of **Apple**:

```
// Use an enum constructor, instance variable, and method.
enum Apple {
    Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

    private int price; // price of each apple

    // Constructor
    Apple(int p) { price = p; }

    int getPrice() { return price; }
}

class EnumDemo3 {
    public static void main(String[] args)
    {
        Apple ap;

        // Display price of Winesap.
        System.out.println("Winesap costs " +
                           Apple.Winesap.getPrice() +
                           " cents.\n");

        // Display all apples and prices.
        System.out.println("All apple prices:");
        for(Apple a : Apple.values())
            System.out.println(a + " costs " + a.getPrice() +
                               " cents.");
    }
}
```

The output is shown here:

```
Winesap costs 15 cents.

All apple prices:
Jonathan costs 10 cents.
GoldenDel costs 9 cents.
RedDel costs 12 cents.
Winesap costs 15 cents.
Cortland costs 8 cents.
```

This version of **Apple** adds three things. The first is the instance variable **price**, which is used to hold the price of each variety of apple. The second is the **Apple** constructor, which is passed the price of an apple. The third is the method **getPrice()**, which returns the value of **price**.

When the variable **ap** is declared in **main()**, the constructor for **Apple** is called once for each constant that is specified. Notice how the arguments to the constructor are specified, by putting them inside parentheses after each constant, as shown here:

```
Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15),  
Cortland(8);
```

These values are passed to the **p** parameter of **Apple()**, which then assigns this value to **price**. Again, the constructor is called once for each constant.

Because each enumeration constant has its own copy of **price**, you can obtain the price of a specified type of apple by calling **getPrice()**. For example, in **main()**, the price of a Winesap is obtained by the following call:

```
Apple.Winesap.getPrice()
```

The prices of all varieties are obtained by cycling through the enumeration using a **for** loop. Because there is a copy of **price** for each enumeration constant, the value associated with one constant is separate and distinct from the value associated with another constant. This is a powerful concept, which is only available when enumerations are implemented as classes, as Java does.

Although the preceding example contains only one constructor, an **enum** can offer two or more overloaded forms, just as can any other class. For example, this version of **Apple** provides a default constructor that initializes the price to **-1**, to indicate that no price data is available:

```
// Use an enum constructor.  
enum Apple {  
    Jonathan(10), GoldenDel(9), RedDel, Winesap(15), Cortland(8);  
  
    private int price; // price of each apple  
  
    // Constructor  
    Apple(int p) { price = p; }  
  
    // Overloaded constructor  
    Apple() { price = -1; }  
  
    int getPrice() { return price; }  
}
```

Notice that in this version, **RedDel** is not given an argument. This means that the default constructor is called, and **RedDel**'s price variable is given the value **-1**.

Here are two restrictions that apply to enumerations. First, an enumeration can't inherit another class. Second, an **enum** cannot be a superclass. This means that an **enum** can't be extended. Otherwise, **enum** acts much like any other class type. The key is to remember that each of the enumeration constants is an object of the class in which it is defined.

Enumerations Inherit Enum

Although you can't inherit a superclass when declaring an **enum**, all enumerations automatically inherit one: **java.lang.Enum**. This class defines several methods that are available for use by all enumerations. The **Enum** class is described in detail in Part II, but three of its methods warrant a discussion at this time.

You can obtain a value that indicates an enumeration constant's position in the list of constants. This is called its *ordinal value*, and it is retrieved by calling the **ordinal()** method, shown here:

```
final int ordinal()
```

It returns the ordinal value of the invoking constant. Ordinal values begin at zero. Thus, in the **Apple** enumeration, **Jonathan** has an ordinal value of 0,

GoldenDel has an ordinal value of 1, **RedDel** has an ordinal value of 2, and so on.

You can compare the ordinal value of two constants of the same enumeration by using the **compareTo()** method. It has this general form:

```
final int compareTo(enum-type e)
```

Here, *enum-type* is the type of the enumeration, and *e* is the constant being compared to the invoking constant. Remember, both the invoking constant and *e* must be of the same enumeration. If the invoking constant has an ordinal value less than *e*'s, then **compareTo()** returns a negative value. If the two ordinal values are the same, then zero is returned. If the invoking constant has an ordinal value greater than *e*'s, then a positive value is returned.

You can compare for equality an enumeration constant with any other object by using **equals()**, which overrides the **equals()** method defined by **Object**. Although **equals()** can compare an enumeration constant to any other object, those two objects will be equal only if they both refer to the same constant, within the same enumeration. Simply having ordinal values in common will not cause **equals()** to return true if the two constants are from different enumerations.

Remember, you can compare two enumeration references for equality by using `= =`.

The following program demonstrates the **ordinal()**, **compareTo()**, and **equals()** methods:

```
// Demonstrate ordinal(), compareTo(), and equals().  
  
// An enumeration of apple varieties.  
enum Apple {  
    Jonathan, GoldenDel, RedDel, Winesap, Cortland  
}  
  
class EnumDemo4 {  
    public static void main(String[] args)  
    {  
        Apple ap, ap2, ap3;  
  
        // Obtain all ordinal values using ordinal().  
        System.out.println("Here are all apple constants" +  
                           " and their ordinal values: ");  
        for(Apple a : Apple.values())  
            System.out.println(a + " " + a.ordinal());  
  
        ap = Apple.RedDel;  
        ap2 = Apple.GoldenDel;  
        ap3 = Apple.RedDel;  
  
        System.out.println();  
  
        // Demonstrate compareTo() and equals()  
        if(ap.compareTo(ap2) < 0)  
            System.out.println(ap + " comes before " + ap2);  
  
        if(ap.compareTo(ap2) > 0)  
            System.out.println(ap2 + " comes before " + ap);  
  
        if(ap.compareTo(ap3) == 0)  
            System.out.println(ap + " equals " + ap3);  
  
        System.out.println();  
  
        if(ap.equals(ap2))  
            System.out.println("Error!");  
  
        if(ap.equals(ap3))  
            System.out.println(ap + " equals " + ap3);
```

```
    if(ap == ap3)
        System.out.println(ap + " == " + ap3);

    }
}
```

The output from the program is shown here:

```
Here are all apple constants and their ordinal values:
Jonathan 0
GoldenDel 1
RedDel 2
Winesap 3
Cortland 4

GoldenDel comes before RedDel
RedDel equals RedDel

RedDel equals RedDel
RedDel == RedDel
```

Another Enumeration Example

Before moving on, we will look at a different example that uses an **enum**. In [Chapter 9](#), we created an automated “decision maker” program. In that version, variables called **NO**, **YES**, **MAYBE**, **LATER**, **SOON**, and **NEVER** were declared within an interface and used to represent the possible answers. While there is nothing technically wrong with that approach, using an enumeration is a better choice. Here is an improved version of that program that uses an **enum** called **Answers** to define the answers. You should compare this version to the original in [Chapter 9](#).

```
// An improved version of the "Decision Maker"
// program from Chapter 9. This version uses an
// enum, rather than interface variables, to
// represent the answers.

import java.util.Random;

// An enumeration of the possible answers.
enum Answers {
    NO, YES, MAYBE, LATER, SOON, NEVER
}

class Question {
    Random rand = new Random();
    Answers ask() {
        int prob = (int) (100 * rand.nextDouble());

        if (prob < 15)
            return Answers.MAYBE; // 15%
        else if (prob < 30)
            return Answers.NO; // 15%
        else if (prob < 60)
            return Answers.YES; // 30%
```

```
        else if (prob < 75)
            return Answers.LATER; // 15%
        else if (prob < 98)
            return Answers.SOON; // 13%
        else
            return Answers.NEVER; // 2%
    }
}

class AskMe {
    static void answer(Answers result) {
        switch(result) {
        case NO:
            System.out.println("No");
            break;
        case YES:
            System.out.println("Yes");
            break;
        case MAYBE:
            System.out.println("Maybe");
            break;
        case LATER:
            System.out.println("Later");
            break;
        case SOON:
            System.out.println("Soon");
            break;
        case NEVER:
            System.out.println("Never");
            break;
        }
    }
}

public static void main(String[] args) {
    Question q = new Question();
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
}
}
```

Type Wrappers

As you know, Java uses primitive types (also called simple types), such as **int** or **double**, to hold the basic data types supported by the language. Primitive types, rather than objects, are used for these quantities for the sake of performance. Using objects for these values would add an unacceptable overhead to even the simplest of calculations. Thus, the primitive types are not part of the object hierarchy, and they do not inherit **Object**.

Despite the performance benefit offered by the primitive types, there are times when you will need an object representation. For example, you can't pass a primitive type by reference to a method. Also, many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types. To handle these (and other) situations, Java provides *type wrappers*, which are classes that encapsulate a primitive type within an object. The type wrapper classes are described in detail in Part II, but they are introduced here because they relate directly to Java's autoboxing feature.

The type wrappers are **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy. Each is briefly examined next.

Character

Character is a wrapper around a **char**. The constructor for **Character** is

`Character(char ch)`

Here, *ch* specifies the character that will be wrapped by the **Character** object being created.

However, beginning with JDK 9, the **Character** constructor was deprecated, and beginning with JDK 16, it has been deprecated for removal. Today, it is strongly recommended that you use the static method **valueOf()** to obtain a **Character** object. It is shown here:

`static Character valueOf(char ch)`

It returns a **Character** object that wraps *ch*.

To obtain the **char** value contained in a **Character** object, call **charValue()**, shown here:

```
char charValue()
```

It returns the encapsulated character.

Boolean

Boolean is a wrapper around **boolean** values. It defines these constructors:

```
Boolean(boolean boolValue)  
Boolean(String boolString)
```

In the first version, *boolValue* must be either **true** or **false**. In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.

However, beginning with JDK 9, the **Boolean** constructors were deprecated, and beginning with JDK 16, they have been deprecated for removal. Today, it is strongly recommended that you use the static method **valueOf()** to obtain a **Boolean** object. It has the two versions shown here:

```
static Boolean valueOf(boolean boolValue)  
static Boolean valueOf(String boolString)
```

Each returns a **Boolean** object that wraps the indicated value.

To obtain a **boolean** value from a **Boolean** object, use **booleanValue()**, shown here:

```
boolean booleanValue()
```

It returns the **boolean** equivalent of the invoking object.

The Numeric Type Wrappers

By far, the most commonly used type wrappers are those that represent numeric values. These are **Byte**, **Short**, **Integer**, **Long**, **Float**, and **Double**.

All of the numeric type wrappers inherit the abstract class **Number**. **Number** declares methods that return the value of an object in each of the different number formats. These methods are shown here:

```
byte byteValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
short shortValue()
```

For example, **doubleValue()** returns the value of an object as a **double**, **floatValue()** returns the value as a **float**, and so on. These methods are implemented by each of the numeric type wrappers.

All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value. For example, here are the constructors defined for **Integer**:

```
Integer(int num)
Integer(String str)
```

If *str* does not contain a valid numeric value, then a **NumberFormatException** is thrown.

However, beginning with JDK 9, the numeric type-wrapper constructors were deprecated, and beginning with JDK 16, they have been deprecated for removal. Today, it is strongly recommended that you use one of the **valueOf()** methods to obtain a wrapper object. The **valueOf()** method is a static member of all of the numeric wrapper classes, and all numeric classes support forms that convert a numeric value or a string into an object. For example, here are two of the forms supported by **Integer**:

```
static Integer valueOf(int val)
static Integer valueOf(String valStr) throws NumberFormatException
```

Here, *val* specifies an integer value and *valStr* specifies a string that represents a properly formatted numeric value in string form. Each returns an **Integer** object that wraps the specified value. Here is an example:

```
Integer iOb = Integer.valueOf(100);
```

After this statement executes, the value 100 is represented by an **Integer** instance. Thus, **iOb** wraps the value 100 within an object. In addition to the forms of **valueOf()** just shown, the integer wrappers, **Byte**, **Short**, **Integer**, and **Long**, also supply a form that lets you specify a radix.

All of the type wrappers override **toString()**. It returns the human-readable form of the value contained within the wrapper. This allows you to output the value by passing a type wrapper object to **println()**, for example, without having to convert it into its primitive type.

The following program demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value:

```
// Demonstrate a type wrapper.
class Wrap {
    public static void main(String[] args) {

        Integer iOb = Integer.valueOf(100);

        int i = iOb.intValue();

        System.out.println(i + " " + iOb); // displays 100 100
    }
}
```

This program wraps the integer value 100 inside an **Integer** object called **iOb**. The program then obtains this value by calling **intValue()** and stores the result in **i**.

The process of encapsulating a value within an object is called *boxing*. Thus, in the program, this line boxes the value 100 into an **Integer**:

```
Integer iOb = Integer.valueOf(100);
```

The process of extracting a value from a type wrapper is called *unboxing*. For example, the program unboxes the value in **iOb** with this statement:

```
int i = iOb.intValue();
```

The same general procedure used by the preceding program to box and unbox values has been available for use since the original version of Java. However, today, Java provides a more streamlined approach, which is described next.

Autoboxing

Modern versions of Java have included two important features: *autoboxing* and *auto-unboxing*. Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object. Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. There is no need to call a method such as **intValue()** or **doubleValue()**.

Autoboxing and auto-unboxing greatly streamline the coding of several algorithms, removing the tedium of manually boxing and unboxing values. They also help prevent errors. Moreover, they are very important to generics, which operate only on objects. Finally, autoboxing makes working with the Collections Framework (described in Part II) much easier.

With autoboxing, it is not necessary to manually construct an object in order to wrap a primitive type. You need only assign that value to a type-wrapper reference. Java automatically constructs the object for you. For example, here is the modern way to construct an **Integer** object that has the value 100:

```
Integer iOb = 100; // autobox an int
```

Notice that the object is not explicitly boxed. Java handles this for you, automatically.

To unbox an object, simply assign that object reference to a primitive-type variable. For example, to unbox **iOb**, you can use this line:

```
int i = iOb; // auto-unbox
```

Java handles the details for you.

Here is the preceding program rewritten to use autoboxing/unboxing:

```
// Demonstrate autoboxing/unboxing.
class AutoBox {
    public static void main(String[] args) {
        Integer iOb = 100; // autobox an int
        int i = iOb; // auto-unbox
        System.out.println(i + " " + iOb); // displays 100 100
    }
}
```

Autoboxing and Methods

In addition to the simple case of assignments, autoboxing automatically occurs whenever a primitive type must be converted into an object; auto-unboxing takes place whenever an object must be converted into a primitive type. Thus, autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method. For example, consider this:

```
// Autoboxing/unboxing takes place with
// method parameters and return values.

class AutoBox2 {
    // Take an Integer parameter and return
    // an int value;
    static int m(Integer v) {
        return v; // auto-unbox to int
    }

    public static void main(String[] args) {
        // Pass an int to m() and assign the return value
        // to an Integer. Here, the argument 100 is autoboxed
        // into an Integer. The return value is also autoboxed
        // into an Integer.
        Integer iOb = m(100);

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

This program displays the following result:

```
100
```

In the program, notice that **m()** specifies an **Integer** parameter and returns an **int** result. Inside **main()**, **m()** is passed the value 100. Because **m()** is expecting an **Integer**, this value is automatically boxed. Then, **m()** returns the **int** equivalent of its argument. This causes **v** to be auto-unboxed. Next, this **int** value is assigned to **iOb** in **main()**, which causes the **int** return value to be autoboxed.

Autoboxing/Unboxing Occurs in Expressions

In general, autoboxing and unboxing take place whenever a conversion into an object or from an object is required. This applies to expressions. Within an expression, a numeric object is automatically unboxed. The outcome of the expression is reboxed, if necessary. For example, consider the following program:

```
// Autoboxing/unboxing occurs inside expressions.

class AutoBox3 {
    public static void main(String[] args) {

        Integer iOb, iOb2;
        int i;

        iOb = 100;
        System.out.println("Original value of iOb: " + iOb);

        // The following automatically unboxes iOb,
        // performs the increment, and then reboxes
        // the result back into iOb.
        ++iOb;
        System.out.println("After ++iOb: " + iOb);

        // Here, iOb is unboxed, the expression is
        // evaluated, and the result is reboxed and
        // assigned to iOb2.
        iOb2 = iOb + (iOb / 3);
        System.out.println("iOb2 after expression: " + iOb2);

        // The same expression is evaluated, but the
        // result is not reboxed.
        i = iOb + (iOb / 3);
        System.out.println("i after expression: " + i);

    }
}
```

The output is shown here:

```
Original value of iOb: 100
After ++iOb: 101
iOb2 after expression: 134
i after expression: 134
```

In the program, pay special attention to this line:

```
++iOb;
```

This causes the value in **iOb** to be incremented. It works like this: **iOb** is unboxed, the value is incremented, and the result is reboxed.

Auto-unboxing also allows you to mix different types of numeric objects in an expression. Once the values are unboxed, the standard type promotions and conversions are applied. For example, the following program is perfectly valid:

```
class AutoBox4 {
    public static void main(String[] args) {

        Integer iOb = 100;
        Double dOb = 98.6;

        dOb = dOb + iOb;
        System.out.println("dOb after expression: " + dOb);
    }
}
```

The output is shown here:

```
dOb after expression: 198.6
```

As you can see, both the **Double** object **dOb** and the **Integer** object **iOb** participated in the addition, and the result was reboxed and stored in **dOb**.

Because of auto-unboxing, you can use **Integer** numeric objects to control a **switch** statement. For example, consider this fragment:

```
Integer iOb = 2;

switch(iOb) {
    case 1: System.out.println("one");
    break;
    case 2: System.out.println("two");
    break;
    default: System.out.println("error");
}
```

When the **switch** expression is evaluated, **iOb** is unboxed and its **int** value is obtained.

As the examples in the programs show, because of autoboxing/unboxing, using numeric objects in an expression is both intuitive and easy. In the early days of Java, such code would have involved casts and calls to methods such as `intValue()`.

Autoboxing/Unboxing Boolean and Character Values

As described earlier, Java also supplies wrappers for `boolean` and `char`. These are **Boolean** and **Character**. Autoboxing/unboxing applies to these wrappers, too. For example, consider the following program:

```
// Autoboxing/unboxing a Boolean and Character.

class AutoBox5 {
    public static void main(String[] args) {

        // Autobox/unbox a boolean.
        Boolean b = true;

        // Below, b is auto-unboxed when used in
        // a conditional expression, such as an if.
        if(b) System.out.println("b is true");

        // Autobox/unbox a char.
        Character ch = 'x'; // box a char
        char ch2 = ch; // unbox a char

        System.out.println("ch2 is " + ch2);
    }
}
```

The output is shown here:

```
b is true
ch2 is x
```

The most important thing to notice about this program is the auto-unboxing of **b** inside the **if** conditional expression. As you should recall, the

conditional expression that controls an **if** must evaluate to type **boolean**. Because of auto-unboxing, the **boolean** value contained within **b** is automatically unboxed when the conditional expression is evaluated. Thus, with autoboxing/unboxing, a **Boolean** object can be used to control an **if** statement.

Because of auto-unboxing, a **Boolean** object can now also be used to control any of Java's loop statements. When a **Boolean** is used as the conditional expression of a **while**, **for**, or **do/while**, it is automatically unboxed into its **boolean** equivalent. For example, this is perfectly valid code:

```
Boolean b;  
// ...  
while(b) { // ...
```

Autoboxing/Unboxing Helps Prevent Errors

In addition to the convenience that it offers, autoboxing/unboxing can also help prevent errors. For example, consider the following program:

```
// An error produced by manual unboxing.  
class UnboxingError {  
    public static void main(String[] args) {  
  
        Integer iOb = 1000; // autobox the value 1000  
  
        int i = iOb.byteValue(); // manually unbox as byte !!!  
  
        System.out.println(i); // does not display 1000 !  
    }  
}
```

This program displays not the expected value of 1000, but –24! The reason is that the value inside **iOb** is manually unboxed by calling **byteValue()**, which causes the truncation of the value stored in **iOb**, which is 1000. This results in the garbage value of –24 being assigned to **i**. Auto-unboxing prevents this type of error because the value in **iOb** will always auto-unbox into a value compatible with **int**.

In general, because autoboxing always creates the proper object, and auto-unboxing always produces the proper value, there is no way for the process to produce the wrong type of object or value. In the rare instances where you want a type different than that produced by the automated process, you can still manually box and unbox values. Of course, the benefits of autoboxing/unboxing are lost. In general, you should employ autoboxing/unboxing. It is the way that modern Java code is written.

A Word of Warning

Because of autoboxing and auto-unboxing, some might be tempted to use objects such as **Integer** or **Double** exclusively, abandoning primitives altogether. For example, with autoboxing/unboxing, it is possible to write code like this:

```
// A bad use of autoboxing/unboxing!
Double a, b, c;

a = 10.0;
b = 4.0;

c = Math.sqrt(a*a + b*b);

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

In this example, objects of type **Double** hold values that are used to calculate the hypotenuse of a right triangle. Although this code is technically correct and does, in fact, work properly, it is a very bad use of autoboxing/unboxing. It is far less efficient than the equivalent code written using the primitive type **double**. The reason is that each autobox and auto-unbox adds overhead that is not present if the primitive type is used.

In general, you should restrict your use of the type wrappers to only those cases in which an object representation of a primitive type is required. Autoboxing/unboxing was not added to Java as a “back door” way of eliminating the primitive types.

Annotations

Java provides a feature that enables you to embed supplemental information into a source file. This information, called an *annotation*, does not change the actions of a program. Thus, an annotation leaves the semantics of a program unchanged. However, this information can be used by various tools during both development and deployment. For example, an annotation might be processed by a source-code generator. The term *metadata* is also used to refer to this feature, but the term *annotation* is the most descriptive and more commonly used.

Annotation Basics

An annotation is created through a mechanism based on the **interface**. Let's begin with an example. Here is the declaration for an annotation called **MyAnno**:

```
// A simple annotation type.  
@interface MyAnno {  
    String str();  
    int val();  
}
```

First, notice the @ that precedes the keyword **interface**. This tells the compiler that an annotation type is being declared. Next, notice the two members **str()** and **val()**. All annotations consist solely of method declarations. However, you don't provide bodies for these methods. Instead, Java implements these methods. Moreover, the methods act much like fields, as you will see.

An annotation cannot include an **extends** clause. However, all annotation types automatically extend the **Annotation** interface. Thus, **Annotation** is a super-interface of all annotations. It is declared within the **java.lang.annotation** package. It overrides **hashCode()**, **equals()**, and **toString()**, which are defined by **Object**. It also specifies **annotationType()**, which returns a **Class** object that represents the invoking annotation.

Once you have declared an annotation, you can use it to annotate something. Initially, annotations could be used only on declarations, and that is where we will begin. (JDK 8 added the ability to annotate type use, and this is described later in this chapter. However, the same basic

techniques apply to both kinds of annotations.) Any type of declaration can have an annotation associated with it. For example, classes, methods, fields, parameters, and **enum** constants can be annotated. Even an annotation can be annotated. In all cases, the annotation precedes the rest of the declaration.

When you apply an annotation, you give values to its members. For example, here is an example of **MyAnno** being applied to a method declaration:

```
// Annotate a method.  
@MyAnno(str = "Annotation Example", val = 100)  
public static void myMeth() { // ...}
```

This annotation is linked with the method **myMeth()**. Look closely at the annotation syntax. The name of the annotation, preceded by an **@**, is followed by a parenthesized list of member initializations. To give a member a value, that member's name is assigned a value. Therefore, in the example, the string "Annotation Example" is assigned to the **str** member of **MyAnno**. Notice that no parentheses follow **str** in this assignment. When an annotation member is given a value, only its name is used. Thus, annotation members look like fields in this context.

Specifying a Retention Policy

Before exploring annotations further, it is necessary to discuss *annotation retention policies*. A retention policy determines at what point an annotation is discarded. Java defines three such policies, which are encapsulated within the **java.lang.annotation.RetentionPolicy** enumeration. They are **SOURCE**, **CLASS**, and **RUNTIME**.

An annotation with a retention policy of **SOURCE** is retained only in the source file and is discarded during compilation.

An annotation with a retention policy of **CLASS** is stored in the **.class** file during compilation. However, it is not available through the JVM during run time.

An annotation with a retention policy of **RUNTIME** is stored in the **.class** file during compilation and is available through the JVM during run time. Thus, **RUNTIME** retention offers the greatest annotation persistence.

NOTE An annotation on a local variable declaration is not retained in the `.class` file.

A retention policy is specified for an annotation by using one of Java's built-in annotations: `@Retention`. Its general form is shown here:

```
@Retention(retention-policy)
```

Here, *retention-policy* must be one of the previously discussed enumeration constants. If no retention policy is specified for an annotation, then the default policy of **CLASS** is used.

The following version of **MyAnno** uses `@Retention` to specify the **RUNTIME** retention policy. Thus, **MyAnno** will be available to the JVM during program execution.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}
```

Obtaining Annotations at Run Time by Use of Reflection

Although annotations are designed mostly for use by other development or deployment tools, if they specify a retention policy of **RUNTIME**, then they can be queried at run time by any Java program through the use of *reflection*. Reflection is the feature that enables information about a class to be obtained at run time. The reflection API is contained in the **java.lang.reflect** package. There are a number of ways to use reflection, and we won't examine them all here. We will, however, walk through a few examples that apply to annotations.

The first step to using reflection is to obtain a **Class** object that represents the class whose annotations you want to obtain. **Class** is one of Java's built-in classes and is defined in **java.lang**. It is described in detail in Part II. There are various ways to obtain a **Class** object. One of the easiest is to call **getClass()**, which is a method defined by **Object**. Its general form is shown here:

```
final Class<?> getClass()
```

It returns the **Class** object that represents the invoking object.

NOTE Notice the <?> that follows **Class** in the declaration of **getClass()** just shown. This is related to Java's generics feature. **getClass()** and several other reflection-related methods discussed in this chapter make use of generics. Generics are described in [Chapter 14](#). However, an understanding of generics is not needed to grasp the fundamental principles of reflection.

After you have obtained a **Class** object, you can use its methods to obtain information about the various items declared by the class, including its annotations. If you want to obtain the annotations associated with a specific item declared within a class, you must first obtain an object that represents that item. For example, **Class** supplies (among others) the **getMethod()**, **getField()**, and **getConstructor()** methods, which obtain information about a method, field, and constructor, respectively. These methods return objects of type **Method**, **Field**, and **Constructor**.

To understand the process, let's work through an example that obtains the annotations associated with a method. To do this, you first obtain a **Class** object that represents the class, and then you call **getMethod()** on that **Class** object, specifying the name of the method. **getMethod()** has this general form:

```
Method getMethod(String methName, Class<?> ... paramTypes)
```

The name of the method is passed in *methName*. If the method has arguments, then **Class** objects representing those types must also be specified by *paramTypes*. Notice that *paramTypes* is a varargs parameter. This means that you can specify as many parameter types as needed, including zero. **getMethod()** returns a **Method** object that represents the method. If the method can't be found, **NoSuchMethodException** is thrown.

From a **Class**, **Method**, **Field**, or **Constructor** object, you can obtain a specific annotation associated with that object by calling **getAnnotation()**. Its general form is shown here:

```
<A extends Annotation> getAnnotation(Class<A> annoType)
```

Here, *annoType* is a **Class** object that represents the annotation in which you are interested. The method returns a reference to the annotation. Using this reference, you can obtain the values associated with the annotation's members. The method returns **null** if the annotation is not found, which will be the case if the annotation does not have **RUNTIME** retention.

Here is a program that assembles all of the pieces shown earlier and uses reflection to display the annotation associated with a method:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// An annotation type declaration.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

class Meta {

    // Annotate a method.
    @MyAnno(str = "Annotation Example", val = 100)
    public static void myMeth() {
        Meta ob = new Meta();

        // Obtain the annotation for this method
        // and display the values of the members.
    }
}
```

```

try {
    // First, get a Class object that represents
    // this class.
    Class<?> c = ob.getClass();

    // Now, get a Method object that represents
    // this method.
    Method m = c.getMethod("myMeth");

    // Next, get the annotation for this class.
    MyAnno anno = m.getAnnotation(MyAnno.class);

    // Finally, display the values.
    System.out.println(anno.str() + " " + anno.val());
} catch (NoSuchMethodException exc) {
    System.out.println("Method Not Found.");
}
}

public static void main(String[] args) {
    myMeth();
}
}

```

The output from the program is shown here:

Annotation Example 100

This program uses reflection as described to obtain and display the values of **str** and **val** in the **MyAnno** annotation associated with **myMeth()** in the **Meta** class. There are two things to pay special attention to. First, in the line

```
MyAnno anno = m.getAnnotation(MyAnno.class);
```

notice the expression **MyAnno.class**. This expression evaluates to a **Class** object of type **MyAnno**, the annotation. This construct is called a *class literal*. You can use this type of expression whenever a **Class** object of a known class is needed. For example, this statement could have been used to obtain the **Class** object for **Meta**:

```
Class<?> c = Meta.class;
```

Of course, this approach only works when you know the class name of an object in advance, which might not always be the case. In general, you can obtain a class literal for classes, interfaces, primitive types, and arrays. (Remember, the `<?>` syntax relates to Java's generics feature. It is described in [Chapter 14](#).)

The second point of interest is the way the values associated with `str` and `val` are obtained when they are output by the following line:

```
System.out.println(anno.str() + " " + anno.val());
```

Notice that they are invoked using the method-call syntax. This same approach is used whenever the value of an annotation member is required.

A Second Reflection Example

In the preceding example, `myMeth()` has no parameters. Thus, when `getMethod()` was called, only the name `myMeth` was passed. However, to obtain a method that has parameters, you must specify class objects representing the types of those parameters as arguments to `getMethod()`. For example, here is a slightly different version of the preceding program:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

class Meta {

    // myMeth now has two arguments.
    @MyAnno(str = "Two Parameters", val = 19)
    public static void myMeth(String str, int i)
    {
        Meta ob = new Meta();

        try {
            Class<?> c = ob.getClass();

            // Here, the parameter types are specified.
            Method m = c.getMethod("myMeth", String.class, int.class);

            MyAnno anno = m.getAnnotation(MyAnno.class);

            System.out.println(anno.str() + " " + anno.val());
        } catch (NoSuchMethodException exc) {
            System.out.println("Method Not Found.");
        }
    }

    public static void main(String[] args) {
        myMeth("test", 10);
    }
}
```

The output from this version is shown here:

Two Parameters 19

In this version, **myMeth()** takes a **String** and an **int** parameter. To obtain information about this method, **getMethod()** must be called as shown here:

```
Method m = c.getMethod("myMeth", String.class, int.class);
```

Here, the **Class** objects representing **String** and **int** are passed as additional arguments.

Obtaining All Annotations

You can obtain all annotations that have **RUNTIME** retention that are associated with an item by calling **getAnnotations()** on that item. It has this general form:

```
Annotation[ ] getAnnotations()
```

It returns an array of the annotations. **getAnnotations()** can be called on objects of type **Class**, **Method**, **Constructor**, and **Field**, among others.

Here is another reflection example that shows how to obtain all annotations associated with a class and with a method. It declares two annotations. It then uses those annotations to annotate a class and a method.

```
// Show all annotations for a class and a method.
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

@Retention(RetentionPolicy.RUNTIME)
@interface What {
    String description();
}

@What(description = "An annotation test class")
@MyAnno(str = "Meta2", val = 99)
class Meta2 {

    @What(description = "An annotation test method")
    @MyAnno(str = "Testing", val = 100)
    public static void myMeth() {
        Meta2 ob = new Meta2();

        try {
            Annotation[] annos = ob.getClass().getAnnotations();

            // Display all annotations for Meta2.
            System.out.println("All annotations for Meta2:");
            for(Annotation a : annos)
                System.out.println(a);

            System.out.println();

            // Display all annotations for myMeth.
            Method m = ob.getClass( ).getMethod("myMeth");
            annos = m.getAnnotations();
        }
    }
}
```

```

        System.out.println("All annotations for myMeth:");
        for(Annotation a : annos)
            System.out.println(a);

    } catch (NoSuchMethodException exc) {
        System.out.println("Method Not Found.");
    }
}

public static void main(String[] args) {
    myMeth();
}
}

```

The output is shown here:

```
All annotations for Meta2:
@What(description=An annotation test class)
@MyAnno(str=Meta2, val=99)
```

```
All annotations for myMeth:
@What(description=An annotation test method)
@MyAnno(str=Testing, val=100)
```

The program uses **getAnnotations()** to obtain an array of all annotations associated with the **Meta2** class and with the **myMeth()** method. As explained, **getAnnotations()** returns an array of **Annotation** objects. Recall that **Annotation** is a super-interface of all annotation interfaces and that it overrides **toString()** in **Object**. Thus, when a reference to an **Annotation** is output, its **toString()** method is called to generate a string that describes the annotation, as the preceding output shows.

The AnnotatedElement Interface

The methods **getAnnotation()** and **getAnnotations()** used by the preceding examples are defined by the **AnnotatedElement** interface, which is defined in **java.lang.reflect**. This interface supports reflection for

annotations and is implemented by the classes **Method**, **Field**, **Constructor**, **Class**, and **Package**, among others.

In addition to **getAnnotation()** and **getAnnotations()**, **AnnotatedElement** defines several other methods. Two have been available since annotations were initially added to Java. The first is **getDeclaredAnnotations()**, which has this general form:

```
Annotation[ ] getDeclaredAnnotations( )
```

It returns all non-inherited annotations present in the invoking object. The second is **isAnnotationPresent()**, which has this general form:

```
default boolean isAnnotationPresent(Class<? extends Annotation>  
annoType)
```

It returns **true** if the annotation specified by *annoType* is associated with the invoking object. It returns **false** otherwise. To these, JDK 8 added **getDeclaredAnnotation()**, **getAnnotationsByType()**, and **getDeclaredAnnotationsByType()**. Of these, the last two automatically work with a repeated annotation.(Repeated annotations are discussed at the end of this chapter.)

Using Default Values

You can give annotation members default values that will be used if no value is specified when the annotation is applied. A default value is specified by adding a **default** clause to a member's declaration. It has this general form:

```
type member( ) default value ;
```

Here, *value* must be of a type compatible with *type*.

Here is **@MyAnno** rewritten to include default values:

```
// An annotation type declaration that includes defaults.  
@Retention(RetentionPolicy.RUNTIME)  
@interface MyAnno {  
    String str() default "Testing";  
    int val() default 9000;  
}
```

This declaration gives a default value of "Testing" to **str** and 9000 to **val**. This means that neither value needs to be specified when **@MyAnno** is used. However, either or both can be given values if desired. Therefore, following are the four ways that **@MyAnno** can be used:

```
@MyAnno() // both str and val default  
@MyAnno(str = "some string") // val defaults  
@MyAnno(val = 100) // str defaults  
@MyAnno(str = "Testing", val = 100) // no defaults
```

The following program demonstrates the use of default values in an annotation:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// An annotation type declaration that includes defaults.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str() default "Testing";
    int val() default 9000;
}

class Meta3 {

    // Annotate a method using the default values.
    @MyAnno()
    public static void myMeth() {
        Meta3 ob = new Meta3();

        // Obtain the annotation for this method
        // and display the values of the members.
        try {
            Class<?> c = ob.getClass();

            Method m = c.getMethod("myMeth");

```



The output is shown here:

```
Testing 9000
```

Marker Annotations

A *marker* annotation is a special kind of annotation that contains no members. Its sole purpose is to mark an item. Thus, its presence as an annotation is sufficient. The best way to determine if a marker annotation is present is to use the method **isAnnotationPresent()**, which is defined by the **AnnotatedElement** interface.

Here is an example that uses a marker annotation. Because a marker interface contains no members, simply determining whether it is present or absent is sufficient.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// A marker annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker { }

class Marker {

    // Annotate a method using a marker.
    // Notice that no ( ) is needed.
    @MyMarker
    public static void myMeth() {
        Marker ob = new Marker();

        try {
            Method m = ob.getClass().getMethod("myMeth");

            // Determine if the annotation is present.
            if(m.isAnnotationPresent(MyMarker.class))
                System.out.println("MyMarker is present.");

        } catch (NoSuchMethodException exc) {
            System.out.println("Method Not Found.");
        }
    }

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

The output, shown here, confirms that **@MyMarker** is present:

```
MyMarker is present.
```

In the program, notice that you do not need to follow **@MyMarker** with parentheses when it is applied. Thus, **@MyMarker** is applied simply by using its name, like this:

`@MyMarker`

It is not wrong to supply an empty set of parentheses, but they are not needed.

Single-Member Annotations

A *single-member* annotation contains only one member. It works like a normal annotation, except that it allows a shorthand form of specifying the value of the member. When only one member is present, you can simply specify the value for that member when the annotation is applied—you don't need to specify the name of the member. However, in order to use this shorthand, the name of the member must be **value**.

Here is an example that creates and uses a single-member annotation:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// A single-member annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MySingle {
    int value(); // this variable name must be value
}

class Single {

    // Annotate a method using a single-member annotation.
    @MySingle(100)
    public static void myMeth() {
        Single ob = new Single();

        try {
            Method m = ob.getClass().getMethod("myMeth");

            MySingle anno = m.getAnnotation(MySingle.class);

            System.out.println(anno.value()); // displays 100
        } catch (NoSuchMethodException exc) {
            System.out.println("Method Not Found.");
        }
    }

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

As expected, this program displays the value 100. In the program, **@MySingle** is used to annotate **myMeth()**, as shown here:

```
@MySingle(100)
```

Notice that **value** = need not be specified.

You can use the single-value syntax when applying an annotation that has other members, but those other members must all have default values. For example, here the value **xyz** is added, with a default value of zero:

```
@interface SomeAnno {  
    int value();  
    int xyz() default 0;  
}
```

In cases in which you want to use the default for **xyz**, you can apply **@SomeAnno**, as shown next, by simply specifying the value of **value** by using the single-member syntax:

```
@SomeAnno (88)
```

In this case, **xyz** defaults to zero, and **value** gets the value 88. Of course, to specify a different value for **xyz** requires that both members be explicitly named, as shown here:

```
@SomeAnno (value = 88, xyz = 99)
```

Remember, whenever you are using a single-member annotation, the name of that member must be **value**.

The Built-In Annotations

Java defines many built-in annotations. Most are specialized, but nine are general purpose. Of these, four are imported from **java.lang.annotation**: **@Retention**, **@Documented**, **@Target**, and **@Inherited**. Five—**@Override**, **@Deprecated**, **@FunctionalInterface**, **@SafeVarargs**, and **@SuppressWarnings**—are included in **java.lang**. Each is described here.

NOTE **java.lang.annotation** also includes the annotations **Repeatable** and **Native**. **Repeatable** supports repeatable annotations, as described later in this chapter. **Native** annotates a field that can be accessed by native code.

@Retention

@Retention is designed to be used only as an annotation to another annotation. It specifies the retention policy as described earlier in this

chapter.

@Documented

The **@Documented** annotation is a marker interface that tells a tool that an annotation is to be documented. It is designed to be used only as an annotation to an annotation declaration.

@Target

The **@Target** annotation specifies the types of items to which an annotation can be applied. It is designed to be used only as an annotation to another annotation. **@Target** takes one argument, which is an array of constants of the **ElementType** enumeration. This argument specifies the types of declarations to which the annotation can be applied. The constants are shown here along with the type of declaration to which they correspond:

Target Constant	Annotation Can Be Applied To
ANNOTATION_TYPE	Another annotation
CONSTRUCTOR	Constructor
FIELD	Field
LOCAL_VARIABLE	Local variable
METHOD	Method
MODULE	Module
PACKAGE	Package
PARAMETER	Parameter
RECORD_COMPONENT	A component of a record (added by JDK 16)
TYPE	Class, interface, or enumeration
TYPE_PARAMETER	Type parameter
TYPE_USE	Type use

You can specify one or more of these values in a **@Target** annotation. To specify multiple values, you must specify them within a braces-

delimited list. For example, to specify that an annotation applies only to fields and local variables, you can use this **@Target** annotation:

```
@Target( { ElementType.FIELD, ElementType.LOCAL_VARIABLE } )
```

If you don't use **@Target**, then the annotation can be used on any declaration. For this reason, it is often a good idea to explicitly specify the target or targets so as to clearly indicate the intended uses of an annotation.

@Inherited

@Inherited is a marker annotation that can be used only on another annotation declaration. Furthermore, it affects only annotations that will be used on class declarations. **@Inherited** causes the annotation for a superclass to be inherited by a subclass. Therefore, when a request for a specific annotation is made to the subclass, if that annotation is not present in the subclass, then its superclass is checked. If that annotation is present in the superclass, and if it is annotated with **@Inherited**, then that annotation will be returned.

@Override

@Override is a marker annotation that can be used only on methods. A method annotated with **@Override** must override a method from a superclass. If it doesn't, a compile-time error will result. It is used to ensure that a superclass method is actually overridden, and not simply overloaded.

@Deprecated

@Deprecated indicates that a declaration is obsolete and not recommended for use. Beginning with JDK 9, **@Deprecated** also allows you to specify the Java version in which the deprecation occurred and whether the deprecated element is slated for removal.

@FunctionalInterface

@FunctionalInterface is a marker annotation designed for use on interfaces. It indicates that the annotated interface is a functional interface. A *functional interface* is an interface that contains one and only one abstract method. Functional interfaces are used by lambda expressions.

(See [Chapter 15](#) for details on functional interfaces and lambda expressions.) If the annotated interface is not a functional interface, a compilation error will be reported. It is important to understand that **@FunctionalInterface** is not needed to create a functional interface. Any interface with exactly one abstract method is, by definition, a functional interface. Thus, **@FunctionalInterface** is purely informational.

@SafeVarargs

@SafeVarargs is a marker annotation that can be applied to methods and constructors. It indicates that no unsafe actions related to a varargs parameter occur. It is used to suppress unchecked warnings on otherwise safe code as it relates to non-reifiable vararg types and parameterized array instantiation. (A non-reifiable type is, essentially, a generic type. Generics are described in [Chapter 14](#).) It must be applied only to vararg methods or constructors. Methods must also be **static**, **final**, or **private**.

@SuppressWarnings

@SuppressWarnings specifies that one or more warnings that might be issued by the compiler are to be suppressed. The warnings to suppress are specified by name, in string form.

Type Annotations

As mentioned earlier, annotations were originally allowed only on declarations. However, for modern versions of Java, annotations can also be specified in most cases in which a type is used. This expanded aspect of annotations is called *type annotation*. For example, you can annotate the return type of a method, the type of **this** within a method, a cast, array levels, an inherited class, and a **throws** clause. You can also annotate generic types, including generic type parameter bounds and generic type arguments. (See [Chapter 14](#) for a discussion of generics.)

Type annotations are important because they enable tools to perform additional checks on code to help prevent errors. Understand that, as a general rule, **javac** will not perform these checks, itself. A separate tool is used for this purpose, although such a tool might operate as a compiler plug-in.

A type annotation must include `ElementType.TYPE_USE` as a target. (Recall that valid annotation targets are specified using the `@Target` annotation, as previously described.) A type annotation applies to the type that the annotation precedes. For example, assuming some type annotation called `@TypeAnno`, the following is legal:

```
void myMeth() throws @TypeAnno NullPointerException { // ...}
```

Here, `@TypeAnno` annotates `NullPointerException` in the `throws` clause.

You can also annotate the type of `this` (called the *receiver*). As you know, `this` is an implicit argument to all instance methods, and it refers to the invoking object. To annotate its type requires the use of another feature that was not originally part of Java. Beginning with JDK 8, you can explicitly declare `this` as the first parameter to a method. In this declaration, the type of `this` must be the type of its class; for example:

```
class SomeClass {
    int myMeth(SomeClass this, int i, int j) { // ...}
```

Here, because `myMeth()` is a method defined by `SomeClass`, the type of `this` is `SomeClass`. Using this declaration, you can now annotate the type of `this`. For example, again assuming that `@TypeAnno` is a type annotation, the following is legal:

```
int myMeth(@TypeAnno SomeClass this, int i, int j) { // ...}
```

It is important to understand that it is not necessary to declare `this` unless you are annotating it. (If `this` is not declared, it is still implicitly passed, as it always has been.) Also, explicitly declaring `this` does not change any aspect of the method's signature because `this` is implicitly declared, by default. Again, you will declare `this` only if you want to apply a type annotation to it. If you do declare `this`, it *must* be the first parameter.

The following program shows a number of the places that a type annotation can be used. It defines several annotations, of which several are for type annotation. The names and targets of the annotations are shown here:

Annotation	Target
@TypeAnno	ElementType.TYPE_USE
@MaxLen	ElementType.TYPE_USE
@NotZeroLen	ElementType.TYPE_USE
@Unique	ElementType.TYPE_USE
@What	ElementType.TYPE_PARAMETER
@EmptyOK	ElementType.FIELD
@Recommended	ElementType.METHOD

Notice that **@EmptyOK**, **@Recommended**, and **@What** are not type annotations. They are included for comparison purposes. Of special interest is **@What**, which is used to annotate a generic type parameter declaration. The comments in the program describe each use.

```
// Demonstrate several type annotations.  
import java.lang.annotation.*;  
import java.lang.reflect.*;  
  
// A marker annotation that can be applied to a type.  
@Target(ElementType.TYPE_USE)  
@interface TypeAnno { }  
  
// Another marker annotation that can be applied to a type.  
@Target(ElementType.TYPE_USE)  
@interface NotZeroLen { }  
  
// Still another marker annotation that can be applied to a type.  
@Target(ElementType.TYPE_USE)  
@interface Unique { }  
  
// A parameterized annotation that can be applied to a type.  
@Target(ElementType.TYPE_USE)  
@interface MaxLen {  
    int value();  
}  
  
// An annotation that can be applied to a type parameter.  
@Target(ElementType.TYPE_PARAMETER)  
@interface What {  
    String description();  
}  
  
// An annotation that can be applied to a field declaration.  
@Target(ElementType.FIELD)  
@interface EmptyOK { }  
  
// An annotation that can be applied to a method declaration.  
@Target(ElementType.METHOD)  
@interface Recommended { }  
  
// Use an annotation on a type parameter.  
class TypeAnnoDemo<@What(description = "Generic data type") T> {  
    // Has a type annotation on a constructor
```

```
// This annotates the field test.  
@EmptyOK String test;  
  
// Use a type annotation to annotate this (the receiver).  
public int f(@TypeAnno TypeAnnoDemo<T> this, int x) {  
    return 10;  
}  
  
// Annotate the return type.  
public @TypeAnno Integer f2(int j, int k) {  
    return j+k;  
}  
  
// Annotate the method declaration.  
public @Recommended Integer f3(String str) {  
    return str.length() / 2;  
}  
  
// Use a type annotation with a throws clause.  
public void f4() throws @TypeAnno NullPointerException {  
    // ...  
}  
  
// Annotate array levels.  
String @MaxLen(10) [] @NotZeroLen [] w;  
  
// Annotate the array element type.  
@TypeAnno Integer[] vec;  
  
public static void myMeth(int i) {  
  
    // Use a type annotation on a type argument.  
    TypeAnnoDemo<@TypeAnno Integer> ob =  
        new TypeAnnoDemo<@TypeAnno Integer>();  
  
    // Use a type annotation with new.  
    @Unique TypeAnnoDemo<Integer> ob2 = new @Unique TypeAnnoDemo<Integer>();  
  
    Object x = Integer.valueOf(10);  
    Integer y;  
  
    // Use a type annotation on a cast
```

Although what most of the annotations in the preceding program refer to is clear, four uses require a bit of discussion. The first is the annotation of a method return type versus the annotation of a method declaration. In the program, pay special attention to these two method declarations:

```
myMeth(10);
// Annotate the return type.
public @TypeAnno Integer f2(int j, int k) {
    return j+k;
}

// Annotate the method declaration.
public @Recommended Integer f3(String str) {
    return str.length() / 2;
}
```

Notice that in both cases, an annotation precedes the method's return type (which is **Integer**). However, the two annotations annotate two different things. In the first case, the **@TypeAnno** annotation annotates **f2()**'s return type. This is because **@TypeAnno** has its target specified as **ElementType.TYPE_USE**, which means that it can be used to annotate type uses. In the second case, **@Recommended** annotates the method declaration itself. This is because **@Recommended** has its target specified as **ElementType.METHOD**. As a result, **@Recommended** applies to the declaration, not the return type. Therefore, the target specification is used to eliminate what, at first glance, appears to be ambiguity between the annotation of a method declaration and the annotation of the method's return type.

One other thing about annotating a method return type: You cannot annotate a return type of **void**.

The second point of interest are the field annotations, shown here:

```
// Annotate the type (in this case String), not the field.
@TypeAnno String str;

// This annotates the field test.
@EmptyOK String test;
```

Here, **@TypeAnno** annotates the type **String**, but **@EmptyOK** annotates the field **test**. Even though both annotations precede the entire declaration,

their targets are different, based on the target element type. If the annotation has the **ElementType.TYPE_USE** target, then the type is annotated. If it has **ElementType.FIELD** as a target, then the field is annotated. Thus, the situation is similar to that just described for methods, and no ambiguity exists. The same mechanism also disambiguates annotations on local variables.

Next, notice how **this** (the receiver) is annotated here:

```
public int f(@TypeAnno TypeAnnoDemo<T> this, int x) {
```

Here, **this** is specified as the first parameter and is of type **TypeAnnoDemo** (which is the class of which **f()** is a member). As explained, an instance method declaration can explicitly specify the **this** parameter for the sake of applying a type annotation.

Finally, look at how array levels are annotated by the following statement:

```
String @MaxLen(10) [] @NotZeroLen [] w;
```

In this declaration, **@MaxLen** annotates the type of the first level, and **@NotZeroLen** annotates the type of the second level. In the declaration

```
@TypeAnno Integer[] vec;
```

the element type **Integer** is annotated.

Repeating Annotations

Beginning with JDK 8, an annotation can be repeated on the same element. This is called *repeating annotations*. For an annotation to be repeatable, it must be annotated with the **@Repeatable** annotation, defined in **java.lang.annotation**. Its **value** field specifies the *container* type for the repeatable annotation. The container is specified as an annotation for which the **value** field is an array of the repeatable annotation type. Thus, to create a repeatable annotation, you must create a container annotation and then specify that annotation type as an argument to the **@Repeatable** annotation.

To access the repeated annotations using a method such as **getAnnotation()**, you will use the container annotation, not the repeatable annotation. The following program shows this approach. It converts the version of **MyAnno** shown previously into a repeatable annotation and demonstrates its use.

```
// Demonstrate a repeated annotation.

import java.lang.annotation.*;
import java.lang.reflect.*;

// Make MyAnno repeatable.
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(MyRepeatedAnnos.class)
@interface MyAnno {
    String str() default "Testing";
    int val() default 9000;
}

// This is the container annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MyRepeatedAnnos {
    MyAnno[] value();
}

class RepeatAnno {

    // Repeat MyAnno on myMeth().
    @MyAnno(str = "First annotation", val = -1)
    @MyAnno(str = "Second annotation", val = 100)
    public static void myMeth(String str, int i)
```

```

{
    RepeatAnno ob = new RepeatAnno();

    try {
        Class<?> c = ob.getClass();

        // Obtain the annotations for myMeth().
        Method m = c.getMethod("myMeth", String.class, int.class);

        // Display the repeated MyAnno annotations.
        Annotation anno = m.getAnnotation(MyRepeatedAnnos.class);
        System.out.println(anno);

    } catch (NoSuchMethodException exc) {
        System.out.println("Method Not Found.");
    }
}

public static void main(String[] args) {
    myMeth("test", 10);
}
}

```

The output is shown here:

```
@MyRepeatedAnnos(value=@MyAnno(val=-1, str="First annotation"),
@MyAnno(val=100, str="Second annotation"))
```

As explained, in order for **MyAnno** to be repeatable, it must be annotated with the **@Repeatable** annotation, which specifies its container annotation. The container annotation is called **MyRepeatedAnnos**. The program accesses the repeated annotations by calling **getAnnotation()**, passing in the class of the container annotation, not the repeatable annotation, itself. As the output shows, the repeated annotations are separated by a comma. They are not returned individually.

Another way to obtain the repeated annotations is to use one of the methods in **AnnotatedElement** that can operate directly on a repeated annotation. These are **getAnnotationsByType()** and

getDeclaredAnnotationsByType(). Here, we will use the former. It is shown here:

```
default <T extends Annotation> T[ ]
getAnnotationsByType(Class<T> annoType)
```

It returns an array of the annotations of *annoType* associated with the invoking object. If no annotations are present, the array will be of zero length. Here is an example. Assuming the preceding program, the following sequence uses **getAnnotationsByType()** to obtain the repeated **MyAnno** annotations:

```
Annotation[] annos = m.getAnnotationsByType(MyAnno.class);
for(Annotation a : annos)
    System.out.println(a);
```

Here, the repeated annotation type, which is **MyAnno**, is passed to **getAnnotationsByType()**. The returned array contains all of the instances of **MyAnno** associated with **myMeth()**, which, in this example, is two. Each repeated annotation can be accessed via its index in the array. In this case, each **MyAnno** annotation is displayed via a for-each loop.

Some Restrictions

There are a number of restrictions that apply to annotation declarations. First, no annotation can inherit another. Second, all methods declared by an annotation must be without parameters. Furthermore, they must return one of the following:

- A primitive type, such as **int** or **double**
- An object of type **String** or **Class**
- An object of an **enum** type
- An object of another annotation type
- An array of a legal type.

Annotations cannot be generic. In other words, they cannot take type parameters. (Generics are described in [Chapter 14](#).) Finally, annotation methods cannot specify a **throws** clause.

CHAPTER

13

I/O, Try-with-Resources, and Other Topics

This chapter introduces one of Java’s most important packages, **java.io**, which supports Java’s basic I/O (input/output) system, including file I/O. Support for I/O comes from Java’s core API libraries, not from language keywords. For this reason, an in-depth discussion of this topic is found in Part II of this book, which examines several of Java’s API packages. Here, the foundation of this important subsystem is introduced so that you can see how it fits into the larger context of the Java programming and execution environment. This chapter also examines the **try-with-resources** statement and several more Java keywords: **transient**, **volatile**, **instanceof**, **native**, **strictfp**, and **assert**. It concludes by discussing static import and describing another use for the **this** keyword.

I/O Basics

As you may have noticed while reading the preceding 12 chapters, not much use has been made of I/O in the sample programs. In fact, aside from **print()** and **println()**, none of the I/O methods have been used significantly. The reason is simple: Most real applications of Java are not text-based, console programs. Rather, they are either graphically oriented programs that rely on one of Java’s graphical user interface (GUI) frameworks, such as Swing, for user interaction, or they are web applications. Although text-based, console programs are excellent as teaching examples, they do not, as a general rule, constitute an important

use for Java in the real world. Also, Java's support for console I/O is limited and somewhat awkward to use—even in simple sample programs. Text-based console I/O is just not that useful in real-world Java programming.

The preceding paragraph notwithstanding, Java does provide strong, flexible support for I/O as it relates to files and networks. Java's I/O system is cohesive and consistent. In fact, once you understand its fundamentals, the rest of the I/O system is easy to master. A general overview of I/O is presented here. A detailed description is found in [Chapters 22](#) and [23](#).

Streams

Java programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to different types of devices. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the `java.io` package.

NOTE In addition to the stream-based I/O defined in `java.io`, Java also provides buffer- and channel-based I/O, which are defined in `java.nio` and its subpackages. They are described in [Chapter 23](#).

Byte Streams and Character Streams

Java defines two types of I/O streams: byte and character. *Byte streams* provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. *Character streams* provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

The original version of Java (Java 1.0) did not include character streams and, thus, all I/O was byte-oriented. Character streams were added by Java 1.1, and certain byte-oriented classes and methods were deprecated.

Although old code that doesn't use character streams is becoming increasingly rare, it may still be encountered from time to time. As a general rule, old code should be updated to take advantage of character streams where appropriate.

One other point: At the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters.

An overview of both byte-oriented streams and character-oriented streams is presented in the following sections.

The Byte Stream Classes

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**. Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connections, and even memory buffers. The non-deprecated byte stream classes in **java.io** are shown in [Table 13-1](#). A few of these classes are discussed later in this section. Others are described in Part II of this book. Remember, to use the stream classes, you must import **java.io**.

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements InputStream
FilterOutputStream	Implements OutputStream
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains print() and println()
PushbackInputStream	Input stream that allows bytes to be returned to the input stream
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Table 13-1 The Non-DDeprecated Byte Stream Classes in `java.io`

The abstract classes **InputStream** and **OutputStream** define several key methods that the other stream classes implement. Two of the most important are **read()** and **write()**, which, respectively, read and write bytes of data. Each has a form that is abstract and must be overridden by derived stream classes.

The Character Stream Classes

Character streams are defined by using two class hierarchies. At the top are two abstract classes: **Reader** and **Writer**. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these. The character stream classes in **java.io** are shown in [Table 13-2](#).

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains <code>print()</code> and <code>println()</code>
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

Table 13-2 The Character Stream I/O Classes in **java.io**

The abstract classes **Reader** and **Writer** define several key methods that the other stream classes implement. Two of the most important methods are **read()** and **write()**, which read and write characters of data, respectively. Each has a form that is abstract and must be overridden by derived stream classes.

The Predefined Streams

As you know, all Java programs automatically import the **java.lang** package. This package defines a class called **System**, which encapsulates several aspects of the run-time environment. For example, using some of its methods, you can obtain the current time and the settings of various properties associated with the system. **System** also contains three predefined stream variables: **in**, **out**, and **err**. These fields are declared as **public**, **static**, and **final** within **System**. This means that they can be used by any other part of your program and without reference to a specific **System** object.

System.out refers to the standard output stream. By default, this is the console. **System.in** refers to standard input, which is the keyboard by default. **System.err** refers to the standard error stream, which also is the console by default. However, these streams may be redirected to any compatible I/O device.

System.in is an object of type **InputStream**; **System.out** and **System.err** are objects of type **PrintStream**. These are byte streams, even though they are typically used to read and write characters from and to the console. As you will see, you can wrap these within character-based streams, if desired.

The preceding chapters have been using **System.out** in their examples. You can use **System.err** in much the same way. As explained in the next section, use of **System.in** is a little more complicated.

Reading Console Input

In the early days of Java, the only way to perform console input was to use a byte stream. Today, using a byte stream to read console input is still often acceptable, such as when used in sample programs. However, for commercial applications, the preferred method of reading console input is

to use a character-oriented stream. This makes your program easier to internationalize and maintain.

In Java, console input is accomplished (either directly or indirectly) by reading from **System.in**. One way to obtain a character-based stream that is attached to the console is to wrap **System.in** in a **BufferedReader**. The **BufferedReader** class supports a buffered input stream. A commonly used constructor is shown here:

`BufferedReader(Reader inputReader)`

Here, *inputReader* is the stream that is linked to the instance of **BufferedReader** that is being created. **Reader** is an abstract class. One of its concrete subclasses is **InputStreamReader**, which converts bytes to characters.

Beginning with JDK 17, the precise way you obtain an **InputStreamReader** linked to **System.in** has changed. In the past, it was common to use the following **InputStreamReader** constructor for this purpose:

`InputStreamReader(InputStream inputStream)`

Because **System.in** refers to an object of type **InputStream**, it can be used for *inputStream*. Thus, the following line of code shows a common approach used in the past for creating a **BufferedReader** connected to the keyboard:

```
BufferedReader br = new BufferedReader(new  
InputStreamReader(System.in));
```

After this statement executes, **br** is a character-based stream that is linked to the console through **System.in**.

However, beginning with JDK 17, it is now recommended to explicitly specify the charset associated with the console when creating the **InputStreamReader**. A *charset* defines the way that bytes are mapped to characters. Normally, when a charset is not specified, the default charset of the JVM is used. However, in the case of the console, the charset used for console input may differ from this default charset. Thus, it is now recommended that this form of **InputStreamReader** constructor be used:

`InputStreamReader(InputStream inputStream, Charset charset)`

For `charset`, use the charset associated with the console. This charset is returned by `charset()`, which is a new method added by JDK 17 to the **Console** class. (See [Chapter 22](#).) You obtain a **Console** object by calling `System.console()`. It returns a reference to the console, or `null` if no console is present. Therefore, today the following sequence shows one way to wrap `System.in` in a **BufferedReader**:

```
Console con = System.console(); // get the console
if(con==null) return; // if no console present, return

BufferedReader br = new
    BufferedReader(new InputStreamReader(System.in, con.charset()));
```

Of course, in cases in which you know that a console will be present, the sequence can be shortened to:

```
BufferedReader br = new
    BufferedReader(new InputStreamReader(System.in,
                                         System.console().charset()));
```

Because a console is (obviously) required to run the examples in this book, this is the form we will use.

One other point: It is also possible to obtain a **Reader** that is already associated with the console by use of the `reader()` method defined by **Console**. However, we will use the **InputStreamReader** approach as just described because it explicitly demonstrates the way that byte streams and character streams can interact.

Reading Characters

To read a character from a **BufferedReader**, use `read()`. The version of `read()` that we will be using is

```
int read() throws IOException
```

Each time that `read()` is called, it reads a character from the input stream and returns it as an integer value. It returns `-1` when an attempt is made to

read at the end of the stream. As you can see, it can throw an **IOException**.

The following program demonstrates **read()** by reading characters from the console until the user types a "q." Notice that any I/O exceptions that might be generated are simply thrown out of **main()**. Such an approach is common when reading from the console in simple sample programs such as those shown in this book, but in more sophisticated applications, you can handle the exceptions explicitly.

```
// Use a BufferedReader to read characters from the console.  
import java.io.*;  
  
class BRRead {  
    public static void main(String[] args) throws IOException  
    {  
        char c;  
        BufferedReader br = new BufferedReader(new  
            InputStreamReader(System.in, System.console().charset()));  
  
        System.out.println("Enter characters, 'q' to quit.");  
        // read characters  
  
        do {  
            c = (char) br.read();  
            System.out.println(c);  
        } while(c != 'q');  
    }  
}
```

Here is a sample run:

```
Enter characters, 'q' to quit.  
123abcq  
1  
2  
3  
a  
b  
c  
q
```

This output may look a little different from what you expected because **System.in** is line buffered, by default. This means that no input is actually passed to the program until you press ENTER. As you can guess, this does not make **read()** particularly valuable for interactive console input.

Reading Strings

To read a string from the keyboard, use the version of **readLine()** that is a member of the **BufferedReader** class. Its general form is shown here:

```
String readLine() throws IOException
```

As you can see, it returns a **String** object.

The following program demonstrates **BufferedReader** and the **readLine()** method; the program reads and displays lines of text until you enter the word "stop":

```
// Read a string from console using a BufferedReader.  
import java.io.*;  
  
class BRReadLines {  
    public static void main(String[] args) throws IOException  
    {  
        // create a BufferedReader using System.in  
        BufferedReader br = new BufferedReader(new  
            InputStreamReader(System.in, System.console().charset()));  
  
        String str;  
        System.out.println("Enter lines of text.");  
        System.out.println("Enter 'stop' to quit.");  
        do {  
            str = br.readLine();  
            System.out.println(str);  
        } while(!str.equals("stop"));  
    }  
}
```

The next example creates a tiny text editor. It creates an array of **String** objects and then reads in lines of text, storing each line in the array. It will

read up to 100 lines or until you enter "stop." It uses a **BufferedReader** to read from the console.

```
// A tiny editor.
import java.io.*;

class TinyEdit {
    public static void main(String[] args) throws IOException
    {
        // create a BufferedReader using System.in
        BufferedReader br = new BufferedReader(new
            InputStreamReader(System.in, System.console().charset()));

        String[] str = new String[100];
        System.out.println("Enter lines of text.");
        System.out.println("Enter 'stop' to quit.");
        for(int i=0; i<100; i++) {
            str[i] = br.readLine();
            if(str[i].equals("stop")) break;
        }
        System.out.println("\nHere is your file:");
        // display the lines
        for(int i=0; i<100; i++) {
            if(str[i].equals("stop")) break;
            System.out.println(str[i]);
        }
    }
}
```

Here is a sample run:

```
Enter lines of text.  
Enter 'stop' to quit.  
This is line one.  
This is line two.  
Java makes working with strings easy.  
Just create String objects.  
stop  
Here is your file:  
This is line one.  
This is line two.  
Java makes working with strings easy.  
Just create String objects.
```

Writing Console Output

Console output is most easily accomplished with **print()** and **println()**, described earlier, which are used in most of the examples in this book. These methods are defined by the class **PrintStream** (which is the type of object referenced by **System.out**). Even though **System.out** is a byte stream, using it for simple program output is still acceptable. However, a character-based alternative is described in the next section.

Because **PrintStream** is an output stream derived from **OutputStream**, it also implements the low-level method **write()**. Thus, **write()** can be used to write to the console. The simplest form of **write()** defined by **PrintStream** is shown here:

```
void write(int byteval)
```

This method writes the byte specified by *byteval*. Although *byteval* is declared as an integer, only the low-order eight bits are written. Here is a short example that uses **write()** to output the character "A" followed by a newline to the screen:

```
// Demonstrate System.out.write().
class WriteDemo {
    public static void main(String[] args) {
        int b;

        b = 'A';
        System.out.write(b);
        System.out.write('\n');
    }
}
```

You will not often use **write()** to perform console output (although doing so might be useful in some situations) because **print()** and **println()** are substantially easier to use.

The PrintWriter Class

Although using **System.out** to write to the console is acceptable, its use is probably best for debugging purposes or for sample programs, such as those found in this book. For real-world programs, the recommended method of writing to the console when using Java is through a **PrintWriter** stream. **PrintWriter** is one of the character-based classes. Using a character-based class for console output makes internationalizing your program easier.

PrintWriter defines several constructors. The one we will use is shown here:

```
PrintWriter(InputStream outputStream, boolean flushingOn)
```

Here, *outputStream* is an object of type **OutputStream**, and *flushingOn* controls whether Java flushes the output stream every time a **println()** method (among others) is called. If *flushingOn* is **true**, flushing automatically takes place. If **false**, flushing is not automatic.

PrintWriter supports the **print()** and **println()** methods. Thus, you can use these methods in the same way as you used them with **System.out**. If an argument is not a simple type, the **PrintWriter** methods call the object's **toString()** method and then display the result.

To write to the console by using a **PrintWriter**, specify **System.out** for the output stream and automatic flushing. For example, this line of code creates a **PrintWriter** that is connected to console output:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

The following application illustrates using a **PrintWriter** to handle console output:

```
// Demonstrate PrintWriter
import java.io.*;

public class PrintWriterDemo {
    public static void main(String[] args) {
        PrintWriter pw = new PrintWriter(System.out, true);

        pw.println("This is a string");
        int i = -7;
        pw.println(i);
        double d = 4.5e-7;
        pw.println(d);
    }
}
```

The output from this program is shown here:

```
This is a string
-7
4.5E-7
```

Remember, there is nothing wrong with using **System.out** to write simple text output to the console when you are learning Java or debugging your programs. However, using a **PrintWriter** makes your real-world applications easier to internationalize. Because no advantage is gained by using a **PrintWriter** in the sample programs shown in this book, we will continue to use **System.out** to write to the console.

Reading and Writing Files

Java provides a number of classes and methods that allow you to read and write files. Before we begin, it is important to state that the topic of file I/O is quite large, and file I/O is examined in detail in Part II. The purpose of this section is to introduce the basic techniques that read from and write to a file. Although byte streams are used, these techniques can be adapted to the character-based streams.

Two of the most often-used stream classes are **FileInputStream** and **FileOutputStream**, which create byte streams linked to files. To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor. Although both classes support additional constructors, the following are the forms that we will be using:

```
FileInputStream(String fileName) throws FileNotFoundException  
FileOutputStream(String fileName) throws FileNotFoundException
```

Here, *fileName* specifies the name of the file that you want to open. When you create an input stream, if the file does not exist, then

FileNotFoundException is thrown. For output streams, if the file cannot be opened or created, then **FileNotFoundException** is thrown.

FileNotFoundException is a subclass of **IOException**. When an output file is opened, any preexisting file by the same name is destroyed.

NOTE In situations in which a security manager is present, several of the file classes, including **FileInputStream** and **FileOutputStream**, will throw a **SecurityException** if a security violation occurs when attempting to open a file. By default, applications run via **java** do not use a security manager. For that reason, the I/O examples in this book do not need to watch for a possible **SecurityException**. However, other types of applications may use the security manager, and file I/O performed by such an application could generate a **SecurityException**. In that case, you will need to appropriately handle this exception. Be aware that JDK 17 deprecates the security manager for removal.

When you are done with a file, you must close it. This is done by calling the **close()** method, which is implemented by both **FileInputStream** and **FileOutputStream**. It is shown here:

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

Closing a file releases the system resources allocated to the file, allowing them to be used by another file. Failure to close a file can result in

“memory leaks” because of unused resources remaining allocated.

NOTE The `close()` method is specified by the **AutoCloseable** interface in **java.lang**.

AutoCloseable is inherited by the **Closeable** interface in **java.io**. Both interfaces are implemented by the stream classes, including **FileInputStream** and **OutputStream**.

Before moving on, it is important to point out that there are two basic approaches that you can use to close a file when you are done with it. The first is the traditional approach, in which `close()` is called explicitly when the file is no longer needed. This is the approach used by all versions of Java prior to JDK 7 and is, therefore, found in all pre-JDK 7 legacy code. The second is to use the **try-with-resources** statement added by JDK 7, which automatically closes a file when it is no longer needed. In this approach, no explicit call to `close()` is executed. Since you may still encounter pre-JDK 7 legacy code, it is important that you know and understand the traditional approach. Furthermore, the traditional approach could still be the best approach in some situations. Therefore, we will begin with it. The automated approach is described in the following section.

To read from a file, you can use a version of `read()` that is defined within **FileInputStream**. The one that we will use is shown here:

```
int read() throws IOException
```

Each time that it is called, it reads a single byte from the file and returns the byte as an integer value. `read()` returns `-1` when an attempt is made to read at the end of the stream. It can throw an **IOException**.

The following program uses `read()` to input and display the contents of a file that contains ASCII text. The name of the file is specified as a command-line argument.

```
/* Display a text file.  
To use this program, specify the name  
of the file that you want to see.  
For example, to see a file called TEST.TXT,  
use the following command line.  
  
    java ShowFile TEST.TXT  
*/  
  
import java.io.*;  
  
class ShowFile {  
    public static void main(String[] args)  
    {  
        int i;  
        FileInputStream fin;
```

```
// First, confirm that a filename has been specified.  
if(args.length != 1) {  
    System.out.println("Usage: ShowFile filename");  
    return;  
}  
  
// Attempt to open the file.  
try {  
    fin = new FileInputStream(args[0]);  
} catch(FileNotFoundException e) {  
    System.out.println("Cannot Open File");  
    return;  
}  
  
// At this point, the file is open and can be read.  
// The following reads characters until EOF is encountered.  
try {  
    do {  
        i = fin.read();  
        if(i != -1) System.out.print((char) i);  
    } while(i != -1);  
} catch(IOException e) {  
    System.out.println("Error Reading File");  
}  
  
// Close the file.  
try {  
    fin.close();  
} catch(IOException e) {  
    System.out.println("Error Closing File");  
}  
}
```

In the program, notice the **try/catch** blocks that handle the I/O errors that might occur. Each I/O operation is monitored for exceptions, and if an exception occurs, it is handled. Be aware that in simple programs or sample code, it is common to see I/O exceptions simply thrown out of **main()**, as was done in the earlier console I/O examples. Also, in some real-world code, it can be helpful to let an exception propagate to a calling routine to

let the caller know that an I/O operation failed. However, most of the file I/O examples in this book handle all I/O exceptions explicitly, as shown, for the sake of illustration.

Although the preceding example closes the file stream after the file is read, there is a variation that is often useful. The variation is to call **close()** within a **finally** block. In this approach, all of the methods that access the file are contained within a **try** block, and the **finally** block is used to close the file. This way, no matter how the **try** block terminates, the file is closed. Assuming the preceding example, here is how the **try** block that reads the file can be recoded:

```
try {
    do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
    } while(i != -1);

} catch(IOException e) {
    System.out.println("Error Reading File");
} finally {
    // Close file on the way out of the try block.
    try {
        fin.close();
    } catch(IOException e) {
        System.out.println("Error Closing File");
    }
}
```

Although not an issue in this case, one advantage to this approach in general is that if the code that accesses a file terminates because of some non-I/O related exception, the file is still closed by the **finally** block.

Sometimes it's easier to wrap the portions of a program that open the file and access the file within a single **try** block (rather than separating the two) and then use a **finally** block to close the file. For example, here is another way to write the **ShowFile** program:

```
/* Display a text file.  
To use this program, specify the name  
of the file that you want to see.  
For example, to see a file called TEST.TXT,  
use the following command line.  
  
java ShowFile TEST.TXT  
  
This variation wraps the code that opens and  
accesses the file within a single try block.  
The file is closed by the finally block.  
*/  
  
import java.io.*;  
  
class ShowFile {  
    public static void main(String[] args)  
    {  
        int i;  
        FileInputStream fin = null;  
  
        // First, confirm that a filename has been specified.  
        if(args.length != 1) {  
            System.out.println("Usage: ShowFile filename");  
            return;  
        }  
  
        // The following code opens a file, reads characters until EOF  
        // is encountered, and then closes the file via a finally block.  
        try {  
            fin = new FileInputStream(args[0]);  
  
            do {  
                i = fin.read();  
                if(i != -1) System.out.print((char) i);  
            } while(i != -1);  
        }  
    }  
}
```

```
    } catch(FileNotFoundException e) {
        System.out.println("File Not Found.");
    } catch(IOException e) {
        System.out.println("An I/O Error Occurred");
    } finally {
        // Close file in all cases.
        try {
            if(fin != null) fin.close();
        } catch(IOException e) {
            System.out.println("Error Closing File");
        }
    }
}
```

In this approach, notice that **fin** is initialized to **null**. Then, in the **finally** block, the file is closed only if **fin** is not **null**. This works because **fin** will be non-**null** only if the file is successfully opened. Thus, **close()** is not called if an exception occurs while opening the file.

It is possible to make the **try/catch** sequence in the preceding example a bit more compact. Because **FileNotFoundException** is a subclass of **IOException**, it need not be caught separately. For example, here is the sequence recoded to eliminate catching **FileNotFoundException**. In this case, the standard exception message, which describes the error, is displayed.

```
try {
    fin = new FileInputStream(args[0]);

    do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
    } while(i != -1);

} catch(IOException e) {
    System.out.println("I/O Error: " + e);
} finally {
    // Close file in all cases.
    try {
        if(fin != null) fin.close();
    } catch(IOException e) {
        System.out.println("Error Closing File");
    }
}
```

In this approach, any error, including an error opening the file, is simply handled by the single **catch** statement. Because of its compactness, this approach is used by many of the I/O examples in this book. Be aware, however, that this approach is not appropriate in cases in which you want to deal separately with a failure to open a file, such as might be caused if a user mistypes a filename. In such a situation, you might want to prompt for the correct name, for example, before entering a **try** block that accesses the file.

To write to a file, you can use the **write()** method defined by **FileOutputStream**. Its simplest form is shown here:

```
void write(int byteval) throws IOException
```

This method writes the byte specified by *byteval* to the file. Although *byteval* is declared as an integer, only the low-order eight bits are written to the file. If an error occurs during writing, an **IOException** is thrown. The next example uses **write()** to copy a file:

```
/* Copy a file.  
To use this program, specify the name  
of the source file and the destination file.  
For example, to copy a file called FIRST.TXT  
to a file called SECOND.TXT, use the following  
command line.  
  
    java CopyFile FIRST.TXT SECOND.TXT  
*/  
  
import java.io.*;  
  
class CopyFile {  
    public static void main(String[] args) throws IOException  
    {  
        int i;  
        FileInputStream fin = null;  
        FileOutputStream fout = null;  
  
        // First, confirm that both files have been specified.  
        if(args.length != 2) {  
            System.out.println("Usage: CopyFile from to");  
            return;  
        }  
  
        // Copy a File.  
        try {  
            // Attempt to open the files.  
            fin = new FileInputStream(args[0]);  
            fout = new FileOutputStream(args[1]);  
  
            do {  
                i = fin.read();  
                if(i != -1) fout.write(i);  
            } while(i != -1);  
  
        } catch(IOException e) {  
            System.out.println("I/O Error: " + e);  
        } finally {  
            try {  
                if(fin != null) fin.close();  
            } catch(IOException e2) {  
            }  
        }  
    }  
}
```

```

        try {
            if(fout != null) fout.close();
        } catch(IOException e2) {
            System.out.println("Error Closing Output File");
        }
    }
}

```

In the program, notice that two separate **try** blocks are used when closing the files. This ensures that both files are closed, even if the call to **fin.close()** throws an exception.

In general, notice that all potential I/O errors are handled in the preceding two programs by the use of exceptions. This differs from some computer languages that use error codes to report file errors. Not only do exceptions make file handling cleaner, but they also enable Java to easily differentiate the end-of-file condition from file errors when input is being performed.

Automatically Closing a File

In the preceding section, the sample programs have made explicit calls to **close()** to close a file once it is no longer needed. As mentioned, this is the way files were closed when using versions of Java prior to JDK 7.

Although this approach is still valid and useful, JDK 7 added a feature that offers another way to manage resources, such as file streams, by automating the closing process. This feature, sometimes referred to as *automatic resource management*, or *ARM* for short, is based on an expanded version of the **try** statement. The principal advantage of automatic resource management is that it prevents situations in which a file (or other resource) is inadvertently not released after it is no longer needed. As explained, forgetting to close a file can result in memory leaks and could lead to other problems.

Automatic resource management is based on an expanded form of the **try** statement. Here is its general form:

```
try (resource-specification) {  
    // use the resource  
}
```

Typically, *resource-specification* is a statement that declares and initializes a resource, such as a file stream. It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the **try** block ends, the resource is automatically released. In the case of a file, this means that the file is automatically closed. (Thus, there is no need to call **close()** explicitly.) Of course, this form of **try** can also include **catch** and **finally** clauses. This form of **try** is called the *try-with-resources* statement.

NOTE Beginning with JDK 9, it is also possible for the resource specification of the **try** to consist of a variable that has been declared and initialized earlier in the program. However, that variable must be *effectively final*, which means that it has not been assigned a new value after being given its initial value.

The **try-with-resources** statement can be used only with those resources that implement the **AutoCloseable** interface defined by **java.lang**. This interface defines the **close()** method. **AutoCloseable** is inherited by the **Closeable** interface in **java.io**. Both interfaces are implemented by the stream classes. Thus, **try-with-resources** can be used when working with streams, including file streams.

As a first example of automatically closing a file, here is a reworked version of the **ShowFile** program that uses it:

```
/* This version of the ShowFile program uses a try-with-resources
   statement to automatically close a file after it is no longer needed.
*/
import java.io.*;

class ShowFile {
    public static void main(String[] args)
    {
        int i;

        // First, confirm that a filename has been specified.
        if(args.length != 1) {
            System.out.println("Usage: ShowFile filename");
            return;
        }

        // The following code uses a try-with-resources statement to open
        // a file and then automatically close it when the try block is left.
        try(FileInputStream fin = new FileInputStream(args[0])) {

            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);

        } catch(FileNotFoundException e) {
            System.out.println("File Not Found.");
        } catch(IOException e) {
            System.out.println("An I/O Error Occurred");
        }
    }
}
```

In the program, pay special attention to how the file is opened within the **try** statement:

```
try(FileInputStream fin = new FileInputStream(args[0])) {
```

Notice how the resource-specification portion of the **try** declares a **FileInputStream** called **fin**, which is then assigned a reference to the file opened by its constructor. Thus, in this version of the program, the variable **fin** is local to the **try** block, being created when the **try** is entered. When the **try** is left, the stream associated with **fin** is automatically closed by an implicit call to **close()**. You don't need to call **close()** explicitly, which means that you can't forget to close the file. This is a key advantage of using **try-with-resources**.

It is important to understand that a resource declared in the **try** statement is implicitly **final**. This means that you can't assign to the resource after it has been created. Also, the scope of the resource is limited to the **try-with-resources** statement.

Before moving on, it is useful to mention that beginning with JDK 10, you can use local variable type inference to specify the type of the resource declared in a **try-with-resources** statement. To do so, specify the type as **var**. When this is done, the type of the resource is inferred from its initializer. For example, the **try** statement in the preceding program can now be written like this:

```
try(var fin = new FileInputStream(args[0])) {
```

Here, **fin** is inferred to be of type **FileInputStream** because that is the type of its initializer. Because a number of readers will be working in Java environments that predate JDK 10, the **try-with-resource** statements in the remainder of this book will not make use of type inference so that the code works for as many readers as possible. Of course, going forward, you should consider using type inference in your own code.

You can manage more than one resource within a single **try** statement. To do so, simply separate each resource specification with a semicolon. The following program shows an example. It reworks the **CopyFile** program shown earlier so that it uses a single **try-with-resources** statement to manage both **fin** and **fout**.

```
/* A version of CopyFile that uses try-with-resources.  
 It demonstrates two resources (in this case files) being  
 managed by a single try statement.  
 */  
  
import java.io.*;  
  
class CopyFile {  
    public static void main(String[] args) throws IOException  
    {  
        int i;  
  
        // First, confirm that both files have been specified.  
        if(args.length != 2) {  
            System.out.println("Usage: CopyFile from to");  
            return;  
        }  
  
        // Open and manage two files via the try statement.  
        try (FileInputStream fin = new FileInputStream(args[0]);  
             FileOutputStream fout = new FileOutputStream(args[1]))  
        {  
  
            do {  
                i = fin.read();  
                if(i != -1) fout.write(i);  
            } while(i != -1);  
  
        } catch(IOException e) {  
            System.out.println("I/O Error: " + e);  
        }  
    }  
}
```

In this program, notice how the input and output files are opened within the **try** block:

```
try (FileInputStream fin = new FileInputStream(args[0]);  
     FileOutputStream fout = new FileOutputStream(args[1]))  
{  
    // ...
```

After this **try** block ends, both **fin** and **fout** will have been closed. If you compare this version of the program to the previous version, you will see that it is much shorter. The ability to streamline source code is a side-benefit of automatic resource management.

There is one other aspect to **try-with-resources** that needs to be mentioned. In general, when a **try** block executes, it is possible that an exception inside the **try** block will lead to another exception that occurs when the resource is closed in a **finally** clause. In the case of a “normal” **try** statement, the original exception is lost, being preempted by the second exception. However, when using **try-with-resources**, the second exception is *suppressed*. It is not, however, lost. Instead, it is added to the list of suppressed exceptions associated with the first exception. The list of suppressed exceptions can be obtained by using the **getSuppressed()** method defined by **Throwable**.

Because of the benefits that the **try-with-resources** statement offers, it will be used by many, but not all, of the sample programs in this edition of this book. Some of the examples will still use the traditional approach to closing a resource. There are several reasons for this. First, you may encounter legacy code that still relies on the traditional approach. It is important that all Java programmers be fully versed in, and comfortable with, the traditional approach when maintaining this older code. Second, it is possible that some programmers will continue to work in a pre-JDK 7 environment for a period of time. In such situations, the expanded form of **try** is not available. Finally, there may be cases in which explicitly closing a resource is more appropriate than the automated approach. For these reasons, some of the examples in this book will continue to use the traditional approach, explicitly calling **close()**. In addition to illustrating the traditional technique, these examples can also be compiled and run by all readers in all environments.

REMEMBER A few examples in this book use the traditional approach to closing files as a means of illustrating this technique, which is widely used in legacy code. However, for new code, you will usually want to use the automated approach supported by the **try-with-resources** statement just described.

The transient and volatile Modifiers

Java defines two interesting type modifiers: **transient** and **volatile**. These modifiers are used to handle somewhat specialized situations.

When an instance variable is declared as **transient**, its value need not persist when an object is stored. For example:

```
class T {  
    transient int a; // will not persist  
    int b; // will persist  
}
```

Here, if an object of type **T** is written to a persistent storage area, the contents of **a** would not be saved, but the contents of **b** would.

The **volatile** modifier tells the compiler that the variable modified by **volatile** can be changed unexpectedly by other parts of your program. One of these situations involves multithreaded programs. In a multithreaded program, sometimes two or more threads share the same variable. For efficiency considerations, each thread can keep its own, private copy of such a shared variable. The real (or *master*) copy of the variable is updated at various times, such as when a **synchronized** method is entered. While this approach works fine, it may be inefficient at times. In some cases, all that really matters is that the master copy of a variable always reflects its current state. To ensure this, simply specify the variable as **volatile**, which tells the compiler that it must always use the master copy of a **volatile** variable (or, at least, always keep any private copies up to date with the master copy, and vice versa). Also, accesses to the shared variable must be executed in the precise order indicated by the program.

Introducing instanceof

Sometimes, knowing the type of an object during run time is useful. For example, you might have one thread of execution that generates various types of objects, and another thread that processes these objects. In this situation, it might be useful for the processing thread to know the type of each object when it receives it. Another situation in which knowledge of an object's type at run time is important involves casting. In Java, an invalid cast causes a run-time error. Many invalid casts can be caught at compile time. However, casts involving class hierarchies can produce invalid casts

that can be detected only at run time. For example, a superclass called A can produce two subclasses, called B and C. Thus, casting a B object into type A or casting a C object into type A is legal, but casting a B object into type C (or vice versa) isn't legal. Because an object of type A can refer to objects of either B or C, how can you know, at run time, what type of object is actually being referred to before attempting the cast to type C? It could be an object of type A, B, or C. If it is an object of type B, a run-time exception will be thrown. Java provides the run-time operator **instanceof** to answer this question.

Before we begin, it is necessary to state that **instanceof** was significantly enhanced by JDK 17 with a powerful new feature based on pattern matching. Here, the traditional form of **instanceof** is introduced. The enhanced form is covered in [Chapter 17](#).

The traditional **instanceof** operator has this general form:

objref instanceof type

Here, *objref* is a reference to an instance of a class, and *type* is a class type. If *objref* is of the specified type or can be cast into the specified type, then the **instanceof** operator evaluates to **true**. Otherwise, its result is **false**. Thus, **instanceof** is the means by which your program can obtain run-time type information about an object.

The following program demonstrates **instanceof**:

```
// Demonstrate instanceof operator.
class A {
    int i, j;
}

class B {
    int i, j;
}

class C extends A {
```

```
        int k;
    }

    class D extends A {
        int k;
    }

    class InstanceOf {
        public static void main(String[] args) {
            A a = new A();
            B b = new B();
            C c = new C();
            D d = new D();
            if(a instanceof A)
                System.out.println("a is instance of A");
            if(b instanceof B)
                System.out.println("b is instance of B");
            if(c instanceof C)
                System.out.println("c is instance of C");
            if(c instanceof A)
                System.out.println("c can be cast to A");

            if(a instanceof C)
                System.out.println("a can be cast to C");

            System.out.println();

            // compare types of derived types
            A ob;

            ob = d; // A reference to d
            System.out.println("ob now refers to d");
            if(ob instanceof D)
                System.out.println("ob is instance of D");

            System.out.println();

            ob = c; // A reference to c
            System.out.println("ob now refers to c");

            if(ob instanceof D)
                System.out.println("ob can be cast to D");
            else
                System.out.println("ob cannot be cast to D");

            if(ob instanceof A)
                System.out.println("ob can be cast to A");
```

The output from this program is shown here:

```
System.out.println();
a is instance of A
b is instance of B      ist to Object
c is instance of C      may be cast to Object");
c can be cast to A

ob now refers to d      may be cast to Object");
ob is instance of D      may be cast to Object");

ob now refers to c      may be cast to Object");
} ob cannot be cast to D
ob can be cast to A

a may be cast to Object
b may be cast to Object
c may be cast to Object
d may be cast to Object
```

The **instanceof** operator isn't needed by most simple programs because, often, you know the type of object with which you are working. However, it can be very useful when you're writing generalized routines that operate on objects of a complex class hierarchy or that are created from code outside your direct control. As you will see, the pattern matching enhancements described in [Chapter 17](#) streamline its use.

strictfp

With the creation of Java 2 several years ago, the floating-point computation model was relaxed slightly. Specifically, the new model did not require the truncation of certain intermediate values that occur during a computation. This prevented overflow or underflow in some cases. By modifying a class, a method, or interface with **strictfp**, you could ensure that floating-point calculations (and thus all truncations) took place precisely as they did in earlier versions of Java. When a class was modified by **strictfp**, all the methods in the class were also modified by **strictfp** automatically. However, beginning with JDK 17, all floating-point computations are now strict, and **strictfp** is obsolete and no longer required. Its use will now generate a warning message.

For versions of Java prior to JDK 17, the following example illustrates **strictfp**. It tells Java to use the original floating-point model for calculations in all methods defined within **MyClass**:

```
strictfp class MyClass { //...
```

Frankly, most programmers never needed to use **strictfp** because it affected only a very small class of problems.

REMEMBER Beginning with JDK 17, **strictfp** has been rendered obsolete and its use will now generate a warning message.

Native Methods

Although it is rare, occasionally you may want to call a subroutine that is written in a language other than Java. Typically, such a subroutine exists as executable code for the CPU and environment in which you are working—that is, native code. For example, you may want to call a native code subroutine to achieve faster execution time. Or, you may want to use a specialized, third-party library, such as a statistical package. However, because Java programs are compiled to bytecode, which is then interpreted (or compiled on the fly) by the Java run-time system, it would seem impossible to call a native code subroutine from within your Java program. Fortunately, this conclusion is false. Java provides the **native** keyword, which is used to declare native code methods. Once declared, these methods can be called from inside your Java program just as you call any other Java method.

To declare a native method, precede the method with the **native** modifier, but do not define any body for the method. For example:

```
public native int meth();
```

After you declare a native method, you must write the native method and follow a rather complex series of steps to link it with your Java code. Consult the Java documentation for current details.

NOTE JDK 21 introduces a more sophisticated API for using code libraries written in languages other than Java code. The new `java.lang.foreign.*` package contains an API that gives greater control over managing memory and calling functions in such libraries.

Using assert

Another interesting keyword is **assert**. It is used during program development to create an *assertion*, which is a condition that should be true during the execution of the program. For example, you might have a method that should always return a positive integer value. You might test this by asserting that the return value is greater than zero using an **assert** statement. At run time, if the condition is true, no other action takes place. However, if the condition is false, then an **AssertionError** is thrown. Assertions are often used during testing to verify that some expected condition is actually met. They are not usually used for released code.

The **assert** keyword has two forms. The first is shown here:

```
assert condition;
```

Here, *condition* is an expression that must evaluate to a Boolean result. If the result is true, then the assertion is true and no other action takes place. If the condition is false, then the assertion fails and a default **AssertionError** object is thrown.

The second form of **assert** is shown here:

```
assert condition: expr ;
```

In this version, *expr* is a value that is passed to the **AssertionError** constructor. This value is converted to its string format and displayed if an assertion fails. Typically, you will specify a string for *expr*, but any non-**void** expression is allowed as long as it defines a reasonable string conversion.

Here is an example that uses **assert**. It verifies that the return value of **getnum()** is positive.

```
// Demonstrate assert.
class AssertDemo {
    static int val = 3;

    // Return an integer.
    static int getnum() {
        return val--;
    }

    public static void main(String[] args)
    {
        int n;

        for(int i=0; i < 10; i++) {
            n = getnum();

            assert n > 0; // will fail when n is 0

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

To enable assertion checking at run time, you must specify the **-ea** option. For example, to enable assertions for **AssertDemo**, execute it using this line:

```
java -ea AssertDemo
```

After compiling and running as just described, the program creates the following output:

```
n is 3
n is 2
n is 1
Exception in thread "main" java.lang.AssertionError
        at AssertDemo.main(AssertDemo.java:17)
```

In **main()**, repeated calls are made to the method **getnum()**, which returns an integer value. The return value of **getnum()** is assigned to **n** and then

tested using this **assert** statement:

```
assert n > 0; // will fail when n is 0
```

This statement will fail when **n** equals 0, which it will after the fourth call. When this happens, an exception is thrown.

As explained, you can specify the message displayed when an assertion fails. For example, if you substitute

```
assert n > 0 : "n is not positive!";
```

for the assertion in the preceding program, then the following output will be generated:

```
n is 3
n is 2

n is 1
Exception in thread "main" java.lang.AssertionError: n is not
positive!
    at AssertDemo.main(AssertDemo.java:17)
```

One important point to understand about assertions is that you must not rely on them to perform any action actually required by the program. The reason is that normally, released code will be run with assertions disabled. For example, consider this variation of the preceding program:

```
// A poor way to use assert!!!
class AssertDemo {
    // get a random number generator
    static int val = 3;

    // Return an integer.
    static int getnum() {
        return val--;
    }

    public static void main(String[] args)
    {
        int n = 0;

        for(int i=0; i < 10; i++) {

            assert (n = getnum()) > 0; // This is not a good idea!

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

In this version of the program, the call to **getnum()** is moved inside the **assert** statement. Although this works fine if assertions are enabled, it will cause a malfunction when assertions are disabled because the call to **getnum()** will never be executed! In fact, **n** must now be initialized because the compiler will recognize that it might not be assigned a value by the **assert** statement.

Assertions can be quite useful because they streamline the type of error checking that is common during development. For example, prior to **assert**, if you wanted to verify that **n** was positive in the preceding program, you had to use a sequence of code similar to this:

```
if(n < 0) {
    System.out.println("n is negative!");
    return; // or throw an exception
}
```

With **assert**, you need only one line of code. Furthermore, you don't have to remove the **assert** statements from your released code.

Assertion Enabling and Disabling Options

When executing code, you can disable all assertions by using the **-da** option. You can enable or disable a specific package (and all of its subpackages) by specifying its name followed by three periods after the **-ea** or **-da** option. For example, to enable assertions in a package called **MyPack**, use

```
-ea:MyPack...
```

To disable assertions in **MyPack**, use

```
-da:MyPack...
```

You can also specify a class with the **-ea** or **-da** option. For example, this enables **AssertDemo** individually:

```
-ea:AssertDemo
```

Static Import

Java includes a feature called *static import* that expands the capabilities of the **import** keyword. By following **import** with the keyword **static**, an **import** statement can be used to import the static members of a class or interface. When using static import, it is possible to refer to static members directly by their names, without having to qualify them with the name of their class. This simplifies and shortens the syntax required to use a static member.

To understand the usefulness of static import, let's begin with an example that does *not* use it. The following program computes the hypotenuse of a right triangle. It uses two static methods from Java's built-in math class **Math**, which is part of **java.lang**. The first is **Math.pow()**, which returns a value raised to a specified power. The second is **Math.sqrt()**, which returns the square root of its argument.

```
// Compute the hypotenuse of a right triangle.  
class Hypot {  
    public static void main(String[] args) {  
        double side1, side2;  
        double hypot;  
        side1 = 3.0;  
        side2 = 4.0;  
  
        // Notice how sqrt() and pow() must be qualified by  
        // their class name, which is Math.  
        hypot = Math.sqrt(Math.pow(side1, 2) +  
                          Math.pow(side2, 2));  
  
        System.out.println("Given sides of lengths " +  
                           side1 + " and " + side2 +  
                           " the hypotenuse is " +  
                           hypot);  
    }  
}
```

Because **pow()** and **sqrt()** are static methods, they must be called through the use of their class's name, **Math**. This results in a somewhat unwieldy hypotenuse calculation:

```
hypot = Math.sqrt(Math.pow(side1, 2) +  
                  Math.pow(side2, 2));
```

As this simple example illustrates, having to specify the class name each time **pow()** or **sqrt()** (or any of Java's other math methods, such as **sin()**, **cos()**, and **tan()**) is used can grow tedious.

You can eliminate the tedium of specifying the class name through the use of static import, as shown in the following version of the preceding program:

```
// Use static import to bring sqrt() and pow() into view.
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

// Compute the hypotenuse of a right triangle.
class Hypot {
    public static void main(String[] args) {
        double side1, side2;
        double hypot;

        side1 = 3.0;
        side2 = 4.0;

        // Here, sqrt() and pow() can be called by themselves,
        // without their class name.
        hypot = sqrt(pow(side1, 2) + pow(side2, 2));

        System.out.println("Given sides of lengths " +
                           side1 + " and " + side2 +
                           " the hypotenuse is " +
                           hypot);
    }
}
```

In this version, the names **sqrt** and **pow** are brought into view by these static import statements:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
```

After these statements, it is no longer necessary to qualify **sqrt()** or **pow()** with their class name. Therefore, the hypotenuse calculation can more conveniently be specified, as shown here:

```
hypot = sqrt(pow(side1, 2) + pow(side2, 2));
```

As you can see, this form is considerably more readable.

There are two general forms of the **import static** statement. The first, which is used by the preceding example, brings into view a single name. Its general form is shown here:

```
import static pkg.type-name.static-member-name ;
```

Here, *type-name* is the name of a class or interface that contains the desired static member. Its full package name is specified by *pkg*. The name of the member is specified by *static-member-name*.

The second form of static import imports all static members of a given class or interface. Its general form is shown here:

```
import static pkg.type-name.*;
```

If you will be using many static methods or fields defined by a class, then this form lets you bring them into view without having to specify each individually. Therefore, the preceding program could have used this single **import** statement to bring both **pow()** and **sqrt()** (and *all other* static members of **Math**) into view:

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

Of course, static import is not limited just to the **Math** class or just to methods. For example, this brings the static field **System.out** into view:

```
import static java.lang.System.out;
```

After this statement, you can output to the console without having to qualify **out** with **System**, as shown here:

```
out.println("After importing System.out, you can use out  
directly.");
```

Whether importing **System.out** as just shown is a good idea is subject to debate. Although it does shorten the statement, it is no longer instantly clear to anyone reading the program that the **out** being referred to is **System.out**.

One other point: in addition to importing the static members of classes and interfaces defined by the Java API, you can also use static import to import the static members of classes and interfaces that you create.

As convenient as static import can be, it is important not to abuse it. Remember, the reason that Java organizes its libraries into packages is to avoid namespace collisions. When you import static members, you are

bringing those members into the current namespace. Thus, you are increasing the potential for namespace conflicts and inadvertent name hiding. If you are using a static member once or twice in the program, it's best not to import it. Also, some static names, such as **System.out**, are so recognizable that you might not want to import them. Static import is designed for those situations in which you are using a static member repeatedly, such as when performing a series of mathematical computations. In essence, you should use, but not abuse, this feature.

Invoking Overloaded Constructors Through **this()**

When working with overloaded constructors, it is sometimes useful for one constructor to invoke another. In Java, this is accomplished by using another form of the **this** keyword. The general form is shown here:

this(*arg-list*)

When **this()** is executed, the overloaded constructor that matches the parameter list specified by *arg-list* is executed first. Then, if there are any statements inside the original constructor, they are executed. The call to **this()** must be the first statement within the constructor.

To understand how **this()** can be used, let's work through a short example. First, consider the following class that *does not* use **this()**:

```
class MyClass {
    int a;
    int b;

    // initialize a and b individually
    MyClass(int i, int j) {
        a = i;
        b = j;
    }

    // initialize a and b to the same value
    MyClass(int i) {
        a = i;
        b = i;
    }

    // give a and b default values of 0
    MyClass( ) {
        a = 0;
        b = 0;
    }
}
```

This class contains three constructors, each of which initializes the values of **a** and **b**. The first is passed individual values for **a** and **b**. The second is passed just one value, which is assigned to both **a** and **b**. The third gives **a** and **b** default values of zero.

By using **this()**, it is possible to rewrite **MyClass** as shown here:

```
class MyClass {
    int a;
    int b;

    // initialize a and b individually
    MyClass(int i, int j) {
        this(i, j);
    }

    MyClass(int i) {
        this(i, i);
    }

    MyClass() {
        this(0, 0);
    }
}
```

```
// initialize a and b to the same value
MyClass(int i) {
    this(i, i); // invokes MyClass(i, i)
}

// give a and b default values of 0
MyClass() {
    this(0); // invokes MyClass(0)
}
}
```

In this version of **MyClass**, the only constructor that actually assigns values to the **a** and **b** fields is **MyClass(int, int)**. The other two constructors simply invoke that constructor (either directly or indirectly) through **this()**. For example, consider what happens when this statement executes:

```
MyClass mc = new MyClass(8);
```

The call to **MyClass(8)** causes **this(8, 8)** to be executed, which translates into a call to **MyClass(8, 8)**, because this is the version of the **MyClass** constructor whose parameter list matches the arguments passed via **this()**. Now, consider the following statement, which uses the default constructor:

```
MyClass mc2 = new MyClass();
```

In this case, **this(0)** is called. This causes **MyClass(0)** to be invoked because it is the constructor with the matching parameter list. Of course, **MyClass(0)** then calls **MyClass(0,0)** as just described.

One reason why invoking overloaded constructors through **this()** can be useful is that it can prevent the unnecessary duplication of code. In many cases, reducing duplicate code decreases the time it takes to load your class because often the object code is smaller. This is especially important for programs delivered via the Internet in which load times are an issue. Using **this()** can also help structure your code when constructors contain a large amount of duplicate code.

However, you need to be careful. Constructors that call **this()** will execute a bit slower than those that contain all of their initialization code inline. This is because the call and return mechanism used when the second

constructor is invoked adds overhead. If your class will be used to create only a handful of objects, or if the constructors in the class that call **this()** will be seldom used, then this decrease in run-time performance is probably insignificant. However, if your class will be used to create a large number of objects (on the order of thousands) during program execution, then the negative impact of the increased overhead could be meaningful. Because object creation affects all users of your class, there will be cases in which you must carefully weigh the benefits of faster load time against the increased time it takes to create an object.

Here is another consideration: for very short constructors, such as those used by **MyClass**, there is often little difference in the size of the object code whether **this()** is used or not. (Actually, there are cases in which no reduction in the size of the object code is achieved.) This is because the bytecode that sets up and returns from the call to **this()** adds instructions to the object file. Therefore, in these types of situations, even though duplicate code is eliminated, using **this()** will not obtain significant savings in terms of load time. However, the added cost in terms of overhead to each object's construction will still be incurred. Therefore, **this()** is most applicable to constructors that contain large amounts of initialization code, not those that simply set the value of a handful of fields.

There are two restrictions you need to keep in mind when using **this()**. First, you cannot use any instance variable of the constructor's class in a call to **this()**. Second, you cannot use **super()** and **this()** in the same constructor because each must be the first statement in the constructor.

A Word About Value-Based Classes

Beginning with JDK 8, Java has included the concept of a *value-based* class, and a number of classes in the Java API have been classified as value-based. Value-based classes are defined by various rules and restrictions. Here are some examples. They must be final, and their instance variables must also be final. If **equals()** determines that two instances of a value-based class are equal, one instance can be used in place of the other. Also, two equal but separately obtained instances of a value-based class may, in fact, be the same object. Very importantly, you should avoid using instances of a value-based class for synchronization. Additional rules and restrictions apply. Furthermore, the definition of value-based classes has

evolved somewhat over time. Consult the Java documentation for the latest details on value-based classes, including which classes in the API library are documented as value-based.

CHAPTER

14

Generics

Since the original 1.0 release in 1995, many new features have been added to Java. One that has had a profound and long-lasting impact is *generics*. Introduced by JDK 5, generics changed Java in two important ways. First, they added a new syntactical element to the language. Second, they caused changes to many of the classes and methods in the core API. Today, generics are an integral part of Java programming, and a solid understanding of this important feature is required. It is examined here in detail.

Through the use of generics, it is possible to create classes, interfaces, and methods that will work in a type-safe manner with various kinds of data. Many algorithms are logically the same no matter what type of data they are being applied to. For example, the mechanism that supports a stack is the same whether that stack is storing items of type **Integer**, **String**, **Object**, or **Thread**. With generics, you can define an algorithm once, independently of any specific type of data, and then apply that algorithm to a wide variety of data types without any additional effort. The expressive power generics added to the language fundamentally changed the way that Java code is written.

Perhaps the one feature of Java that was most significantly affected by generics is the *Collections Framework*. The Collections Framework is part of the Java API and is described in detail in [Chapter 20](#), but a brief mention is useful now. A *collection* is a group of objects. The Collections Framework defines several classes, such as lists and maps, that manage collections. The collection classes had always been able to work with any type of object. The benefit that generics added is the ability to use the

collection classes with complete type safety. Thus, in addition to being a powerful language element on its own, generics also enabled an existing feature to be substantially improved. This is another reason why generics were such an important addition to Java.

This chapter describes the syntax, theory, and use of generics. It also shows how generics provide type safety for some previously difficult cases. Once you have completed this chapter, you will want to examine [Chapter 20](#), which covers the Collections Framework. There you will find many examples of generics at work.

What Are Generics?

At its core, the term *generics* means *parameterized types*. Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter. Using generics, it is possible to create a single class, for example, that automatically works with different types of data. A class, interface, or method that operates on a parameterized type is called *generic*, as in *generic class* or *generic method*.

It is important to understand that Java has always given you the ability to create generalized classes, interfaces, and methods by operating through references of type **Object**. Because **Object** is the superclass of all other classes, an **Object** reference can refer to any type object. Thus, in pre-generics code, generalized classes, interfaces, and methods used **Object** references to operate on various types of objects. The problem was that they could not do so with type safety.

Generics added the type safety that was lacking. They also streamlined the process, because it is no longer necessary to explicitly employ casts to translate between **Object** and the type of data that is actually being operated upon. With generics, all casts are automatic and implicit. Thus, generics expanded your ability to reuse code and let you do so safely and easily.

CAUTION A Warning to C++ Programmers: Although generics are similar to templates in C++, they are not the same. There are some fundamental differences between the two approaches to generic types. If you have a background in C++, it is important not to jump to conclusions about how generics work in Java.

A Simple Generics Example

Let's begin with a simple example of a generic class. The following program defines two classes. The first is the generic class **Gen**, and the second is **GenDemo**, which uses **Gen**.

```
// A simple generic class.  
// Here, T is a type parameter that  
// will be replaced by a real type  
// when an object of type Gen is created.  
class Gen<T> {  
    T ob; // declare an object of type T  
  
    // Pass the constructor a reference to  
    // an object of type T.  
    Gen(T o) {  
        ob = o;  
    }  
  
    // Return ob.  
    T getOb() {  
        return ob;  
    }  
  
    // Show type of T.
```

```
void showType() {
    System.out.println("Type of T is " +
                       ob.getClass().getName());
}

// Demonstrate the generic class.
class GenDemo {
    public static void main(String[] args) {
        // Create a Gen reference for Integers.
        Gen<Integer> iOb;

        // Create a Gen<Integer> object and assign its
        // reference to iOb. Notice the use of autoboxing
        // to encapsulate the value 88 within an Integer object.
        iOb = new Gen<Integer>(88);

        // Show the type of data used by iOb.
        iOb.showType();

        // Get the value in iOb. Notice that
        // no cast is needed.
        int v = iOb.getOb();
        System.out.println("value: " + v);

        System.out.println();

        // Create a Gen object for Strings.
        Gen<String> strOb = new Gen<String> ("Generics Test");

        // Show the type of data used by strOb.
        strOb.showType();

        // Get the value of strOb. Again, notice
        // that no cast is needed.
        String str = strOb.getOb();
        System.out.println("value: " + str);
    }
}
```

The output produced by the program is shown here:

```
Type of T is java.lang.Integer
value: 88
```

```
Type of T is java.lang.String
value: Generics Test
```

Let's examine this program carefully.

First, notice how **Gen** is declared by the following line:

```
class Gen<T> {
```

Here, **T** is the name of a *type parameter*. This name is used as a placeholder for the actual type that will be passed to **Gen** when an object is created. Thus, **T** is used within **Gen** whenever the type parameter is needed. Notice that **T** is contained within **<>**. This syntax can be generalized. Whenever a type parameter is being declared, it is specified within angle brackets. Because **Gen** uses a type parameter, **Gen** is a generic class, which is also called a *parameterized type*.

In the declaration of **Gen**, there is no special significance to the name **T**. Any valid identifier could have been used, but **T** is traditional. Furthermore, it is recommended that type parameter names be single-character capital letters. Other commonly used type parameter names are **V** and **E**. One other point about type parameter names: Beginning with JDK 10, you cannot use **var** as the name of a type parameter.

Next, **T** is used to declare an object called **ob**, as shown here:

```
T ob; // declare an object of type T
```

As explained, **T** is a placeholder for the actual type that will be specified when a **Gen** object is created. Thus, **ob** will be an object of the type passed to **T**. For example, if type **String** is passed to **T**, then in that instance, **ob** will be of type **String**.

Now consider **Gen**'s constructor:

```
Gen(T o) {
    ob = o;
}
```

Notice that its parameter, **o**, is of type **T**. This means that the actual type of **o** is determined by the type passed to **T** when a **Gen** object is created. Also, because both the parameter **o** and the member variable **ob** are of type **T**, they will both be of the same actual type when a **Gen** object is created.

The type parameter **T** can also be used to specify the return type of a method, as is the case with the **getOb()** method, shown here:

```
T getOb() {  
    return ob;  
}
```

Because **ob** is also of type **T**, its type is compatible with the return type specified by **getOb()**.

The **showType()** method displays the type of **T** by calling **getName()** on the **Class** object returned by the call to **getClass()** on **ob**. The **getClass()** method is defined by **Object** and is thus a member of all class types. It returns a **Class** object that corresponds to the type of the class of the object on which it is called. **Class** defines the **getName()** method, which returns a string representation of the class name.

The **GenDemo** class demonstrates the generic **Gen** class. It first creates a version of **Gen** for integers, as shown here:

```
Gen<Integer> iOb;
```

Look closely at this declaration. First, notice that the type **Integer** is specified within the angle brackets after **Gen**. In this case, **Integer** is a *type argument* that is passed to **Gen**'s type parameter, **T**. This effectively creates a version of **Gen** in which all references to **T** are translated into references to **Integer**. Thus, for this declaration, **ob** is of type **Integer**, and the return type of **getOb()** is of type **Integer**.

Before moving on, it's necessary to state that the Java compiler does not actually create different versions of **Gen**, or of any other generic class. Although it's helpful to think in these terms, it is not what actually happens. Instead, the compiler removes all generic type information, substituting the necessary casts, to make your code *behave as if* a specific version of **Gen** were created. Thus, there is really only one version of **Gen** that actually

exists in your program. The process of removing generic type information is called *erasure*, and we will return to this topic later in this chapter.

The next line assigns to **iOb** a reference to an instance of an **Integer** version of the **Gen** class:

```
iOb = new Gen<Integer>(88);
```

Notice that when the **Gen** constructor is called, the type argument **Integer** is also specified. This is because the type of the object (in this case **iOb**) to which the reference is being assigned is of type **Gen<Integer>**. Thus, the reference returned by **new** must also be of type **Gen<Integer>**. If it isn't, a compile-time error will result. For example, the following assignment will cause a compile-time error:

```
iOb = new Gen<Double>(88.0); // Error!
```

Because **iOb** is of type **Gen<Integer>**, it can't be used to refer to an object of **Gen<Double>**. This type checking is one of the main benefits of generics because it ensures type safety.

NOTE As you will see later in this chapter, it is possible to shorten the syntax used to create an instance of a generic class. In the interest of clarity, we will use the full syntax at this time.

As the comments in the program state, the assignment

```
iOb = new Gen<Integer>(88);
```

makes use of autoboxing to encapsulate the value 88, which is an **int**, into an **Integer**. This works because **Gen<Integer>** creates a constructor that takes an **Integer** argument. Because an **Integer** is expected, Java will automatically box 88 inside one. Of course, the assignment could also have been written explicitly, like this:

```
iOb = new Gen<Integer>(Integer.valueOf(88));
```

However, there would be no benefit to using this version.

The program then displays the type of **ob** within **iOb**, which is **Integer**. Next, the program obtains the value of **ob** by use of the following line:

```
int v = iOb.getOb();
```

Because the return type of **getOb()** is **T**, which was replaced by **Integer** when **iOb** was declared, the return type of **getOb()** is also **Integer**, which unboxes into **int** when assigned to **v** (which is an **int**). Thus, there is no need to cast the return type of **getOb()** to **Integer**. Of course, it's not necessary to use the auto-unboxing feature. The preceding line could have been written like this, too:

```
int v = iOb.getOb().intValue();
```

However, the auto-unboxing feature makes the code more compact.

Next, **GenDemo** declares an object of type **Gen<String>**:

```
Gen<String> strOb = new Gen<String>("Generics Test");
```

Because the type argument is **String**, **String** is substituted for **T** inside **Gen**. This creates (conceptually) a **String** version of **Gen**, as the remaining lines in the program demonstrate.

Generics Work Only with Reference Types

When declaring an instance of a generic type, the type argument passed to the type parameter must be a reference type. You cannot use a primitive type, such as **int** or **char**. For example, with **Gen**, it is possible to pass any class type to **T**, but you cannot pass a primitive type to a type parameter. Therefore, the following declaration is illegal:

```
Gen<int> intOb = new Gen<int>(53); // Error, can't use  
primitive type
```

Of course, not being able to specify a primitive type is not a serious restriction because you can use the type wrappers (as the preceding example did) to encapsulate a primitive type. Further, Java's autoboxing and auto-unboxing mechanism makes the use of the type wrapper transparent.

Generic Types Differ Based on Their Type Arguments

A key point to understand about generic types is that a reference of one specific version of a generic type is not type compatible with another version of the same generic type. For example, assuming the program just shown, the following line of code is in error and will not compile:

```
iOb = strOb; // Wrong!
```

Even though both **iOb** and **strOb** are of type **Gen<T>**, they are references to different types because their type arguments differ. This is part of the way that generics add type safety and prevent errors.

How Generics Improve Type Safety

At this point, you might be asking yourself the following question: Given that the same functionality found in the generic **Gen** class can be achieved without generics, by simply specifying **Object** as the data type and employing the proper casts, what is the benefit of making **Gen** generic? The answer is that generics automatically ensure the type safety of all operations involving **Gen**. In the process, they eliminate the need for you to enter casts and to type-check code by hand.

To understand the benefits of generics, first consider the following program that creates a non-generic equivalent of **Gen**:

```
// NonGen is functionally equivalent to Gen
// but does not use generics.
class NonGen {
    Object ob; // ob is now of type Object

    // Pass the constructor a reference to
    // an object of type Object
    NonGen(Object o) {
        ob = o;
    }

    // Return type Object.
    Object getOb() {
        return ob;
    }

    // Show type of ob.
    void showType() {
        System.out.println("Type of ob is " +
                           ob.getClass().getName());
    }
}

// Demonstrate the non-generic class.
class NonGenDemo {
    public static void main(String[] args) {
        NonGen iOb;

        // Create NonGen Object and store
        // an Integer in it. Autoboxing still occurs.
        iOb = new NonGen(88);

        // Show the type of data used by iOb.
        iOb.showType();

        // Get the value of iOb.
        // This time, a cast is necessary.
        int v = (Integer) iOb.getOb();
        System.out.println("value: " + v);

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

```

        String str = (String) strOb.getOb();
        System.out.println("value: " + str);

        // This compiles, but is conceptually wrong!
        iOb = strOb;
        v = (Integer) iOb.getOb(); // run-time error!
    }
}

// Get the value of strOb.
// Again, notice that a cast is necessary.

```

There are several things of interest in this version. First, notice that **NonGen** replaces all uses of **T** with **Object**. This makes **NonGen** able to store any type of object, as can the generic version. However, it also prevents the Java compiler from having any real knowledge about the type of data actually stored in **NonGen**, which is bad for two reasons. First, explicit casts must be employed to retrieve the stored data. Second, many kinds of type mismatch errors cannot be found until run time. Let's look closely at each problem.

Notice this line:

```
int v = (Integer) iOb.getOb();
```

Because the return type of **getOb()** is **Object**, the cast to **Integer** is necessary to enable that value to be auto-unboxed and stored in **v**. If you remove the cast, the program will not compile. With the generic version, this cast was implicit. In the non-generic version, the cast must be explicit. This is not only an inconvenience, but also a potential source of error.

Now, consider the following sequence from near the end of the program:

```

// This compiles, but is conceptually wrong!
iOb = strOb;
v = (Integer) iOb.getOb(); // run-time error!

```

Here, **strOb** is assigned to **iOb**. However, **strOb** refers to an object that contains a string, not an integer. This assignment is syntactically valid because all **NonGen** references are the same, and any **NonGen** reference can refer to any other **NonGen** object. However, the statement is semantically wrong, as the next line shows. Here, the return type of **getOb()** is cast to **Integer**, and then an attempt is made to assign this value to **v**. The trouble is that **iOb** now refers to an object that stores a **String**, not an

Integer. Unfortunately, without the use of generics, the Java compiler has no way to know this. Instead, a run-time exception occurs when the cast to **Integer** is attempted. As you know, it is extremely bad to have run-time exceptions occur in your code!

The preceding sequence can't occur when generics are used. If this sequence were attempted in the generic version of the program, the compiler would catch it and report an error, thus preventing a serious bug that results in a run-time exception. The ability to create type-safe code in which type-mismatch errors are caught at compile time is a key advantage of generics. Although using **Object** references to create "generic" code has always been possible, that code was not type safe, and its misuse could result in run-time exceptions. Generics prevent this from occurring. In essence, through generics, run-time errors are converted into compile-time errors. This is a major advantage.

A Generic Class with Two Type Parameters

You can declare more than one type parameter in a generic type. To specify two or more type parameters, simply use a comma-separated list. For example, the following **TwoGen** class is a variation of the **Gen** class that has two type parameters:

```
// A simple generic class with two type
// parameters: T and V.
class TwoGen<T, V> {
    T ob1;
    V ob2;

    // Pass the constructor a reference to
    // an object of type T and an object of type V.
    TwoGen(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }

    // Show types of T and V.
    void showTypes() {
        System.out.println("Type of T is " +
                           ob1.getClass().getName());

        System.out.println("Type of V is " +
                           ob2.getClass().getName());
    }

    T getOb1() {
        return ob1;
    }

    V getOb2() {
        return ob2;
    }
}

// Demonstrate TwoGen.
class SimpGen {
    public static void main(String[] args) {

        TwoGen<Integer, String> tgObj =
            new TwoGen<Integer, String>(88, "Generics");

        // Show the types.
        tgObj.showTypes();

        // Obtain and show values.
        ... ...
    }
}
```

```
    T t = tObj.getObj();
    System.out.println("Value: " + v);
}
Type of T is java.lang.Integer
Type of V is java.lang.String
value: 88
value: Generics
```

Notice how **TwoGen** is declared:

```
class TwoGen<T, V> {
```

It specifies two type parameters: **T** and **V**, separated by a comma. Because it has two type parameters, two type arguments must be passed to **TwoGen** when an object is created, as shown next:

```
TwoGen<Integer, String> tgObj =
    new TwoGen<Integer, String>(88, "Generics");
```

In this case, **Integer** is substituted for **T**, and **String** is substituted for **V**.

Although the two type arguments differ in this example, it is possible for both types to be the same. For example, the following line of code is valid:

```
TwoGen<String, String> x = new TwoGen<String, String> ("A",
    "B");
```

In this case, both **T** and **V** would be of type **String**. Of course, if the type arguments were always the same, then two type parameters would be unnecessary.

The General Form of a Generic Class

The generics syntax shown in the preceding examples can be generalized. Here is the syntax for declaring a generic class:

```
class class-name<type-param-list> { // ...
```

Here is the full syntax for declaring a reference to a generic class and instance creation:

```
class-name<type-arg-list> var-name =  
    new class-name<type-arg-list>(cons-arg-list);
```

Bounded Types

In the preceding examples, the type parameters could be replaced by any class type. This is fine for many purposes, but sometimes it is useful to limit the types that can be passed to a type parameter. For example, assume that you want to create a generic class that contains a method that returns the average of an array of numbers. Furthermore, you want to use the class to obtain the average of an array of any type of number, including integers, **floats**, and **doubles**. Thus, you want to specify the type of the numbers generically, using a type parameter. To create such a class, you might try something like this:

```
// Stats attempts (unsuccessfully) to  
// create a generic class that can compute
```

```

// the average of an array of numbers of
// any given type.
//
// The class contains an error!
class Stats<T> {
    T[] nums; // nums is an array of type T

    // Pass the constructor a reference to
    // an array of type T.
    Stats(T[] o) {
        nums = o;
    }

    // Return type double in all cases.
    double average() {
        double sum = 0.0;
        for(int i=0; i < nums.length; i++)
            sum += nums[i].doubleValue(); // Error!!!

        return sum / nums.length;
    }
}

```

In **Stats**, the **average()** method attempts to obtain the **double** version of each number in the **nums** array by calling **doubleValue()**. Because all numeric classes, such as **Integer** and **Double**, are subclasses of **Number**, and **Number** defines the **doubleValue()** method, this method is available to all numeric wrapper classes. The trouble is that the compiler has no way to know that you are intending to create **Stats** objects using only numeric types. Thus, when you try to compile **Stats**, an error is reported that indicates that the **doubleValue()** method is unknown. To solve this problem, you need some way to tell the compiler that you intend to pass only numeric types to **T**. Furthermore, you need some way to *ensure* that *only* numeric types are actually passed.

To handle such situations, Java provides *bounded types*. When specifying a type parameter, you can create an upper bound that declares the superclass from which all type arguments must be derived. This is accomplished through the use of an **extends** clause when specifying the type parameter, as shown here:

<T extends superclass>

This specifies that *T* can only be replaced by *superclass*, or subclasses of *superclass*. Thus, *superclass* defines an inclusive, upper limit.

You can use an upper bound to fix the **Stats** class shown earlier by specifying **Number** as an upper bound, as shown here:

```
// In this version of Stats, the type argument for
// T must be either Number, or a class derived
// from Number.
class Stats<T extends Number> {
    T[] nums; // array of Number or subclass
```

```
// Pass the constructor a reference to
// an array of type Number or subclass.
Stats(T[] o) {
    nums = o;
}

// Return type double in all cases.
double average() {
    double sum = 0.0;

    for(int i=0; i < nums.length; i++)
        sum += nums[i].doubleValue();

    return sum / nums.length;
}

// Demonstrate Stats.
class BoundsDemo {
    public static void main(String[] args) {

        Integer[] inums = { 1, 2, 3, 4, 5 };
        Stats<Integer> iob = new Stats<Integer>(inums);
        double v = iob.average();
        System.out.println("iob average is " + v);

        Double[] dnums = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        Stats<Double> dob = new Stats<Double>(dnums);
        double w = dob.average();
        System.out.println("dob average is " + w);

        // This won't compile because String is not a
        // subclass of Number.
        // String[] strs = { "1", "2", "3", "4", "5" };
        // Stats<String> strob = new Stats<String>(strs);

        // double x = strob.average();
        // System.out.println("strob average is " + v);
    }
}
```

The output is shown here:

```
Average is 3.0  
Average is 3.3
```

Notice how **Stats** is now declared by this line:

```
class Stats<T extends Number> {
```

Because the type **T** is now bounded by **Number**, the Java compiler knows that all objects of type **T** can call **doubleValue()** because it is a method declared by **Number**. This is, by itself, a major advantage. However, as an added bonus, the bounding of **T** also prevents nonnumeric **Stats** objects from being created. For example, if you try removing the comments from the lines at the end of the program, and then try recompiling, you will receive compile-time errors because **String** is not a subclass of **Number**.

In addition to using a class type as a bound, you can also use an interface type. In fact, you can specify multiple interfaces as bounds. Furthermore, a bound can include both a class type and one or more interfaces. In this case, the class type must be specified first. When a bound includes an interface type, only type arguments that implement that interface are legal. When specifying a bound that has a class and an interface, or multiple interfaces, use the **&** operator to connect them. This creates an *intersection type*. For example:

```
class Gen<T extends MyClass & MyInterface> { // ...
```

Here, **T** is bounded by a class called **MyClass** and an interface called **MyInterface**. Thus, any type argument passed to **T** must be a subclass of **MyClass** and implement **MyInterface**. As a point of interest, you can also use a type intersection in a cast.

Using Wildcard Arguments

As useful as type safety is, sometimes it can get in the way of perfectly acceptable constructs. For example, given the **Stats** class shown at the end of the preceding section, assume that you want to add a method called **isSameAvg()** that determines if two **Stats** objects contain arrays that yield

the same average, no matter what type of numeric data each object holds. For example, if one object contains the **double** values 1.0, 2.0, and 3.0, and the other object contains the integer values 2, 1, and 3, then the averages will be the same. One way to implement **isSameAvg()** is to pass it a **Stats** argument, and then compare the average of that argument against the invoking object, returning true only if the averages are the same. For example, you want to be able to call **isSameAvg()**, as shown here:

```
Integer[] inums = { 1, 2, 3, 4, 5 };
Double[] dnums = { 1.1, 2.2, 3.3, 4.4, 5.5 };

Stats<Integer> iob = new Stats<Integer>(inums);
Stats<Double> dob = new Stats<Double>(dnums);

if(iob.isSameAvg(dob))
    System.out.println("Averages are the same.");
else
    System.out.println("Averages differ.");
```

At first, creating **isSameAvg()** seems like an easy problem. Because **Stats** is generic and its **average()** method can work on any type of **Stats** object, it seems that creating **isSameAvg()** would be straightforward. Unfortunately, trouble starts as soon as you try to declare a parameter of type **Stats**. Because **Stats** is a parameterized type, what do you specify for **Stats'** type parameter when you declare a parameter of that type?

At first, you might think of a solution like this, in which **T** is used as the type parameter:

```
// This won't work!
// Determine if two averages are the same.
boolean isSameAvg(Stats<T> ob) {
    if(average() == ob.average())
        return true;

    return false;
}
```

The trouble with this attempt is that it will work only with other **Stats** objects whose type is the same as the invoking object. For example, if the

invoking object is of type **Stats<Integer>**, then the parameter **ob** must also be of type **Stats<Integer>**. It can't be used to compare the average of an object of type **Stats<Double>** with the average of an object of type **Stats<Short>**, for example. Therefore, this approach won't work except in a very narrow context and does not yield a general (that is, generic) solution.

To create a generic **isSameAvg()** method, you must use another feature of Java generics: the *wildcard* argument. The wildcard argument is specified by the ?, and it represents an unknown type. Using a wildcard, here is one way to write the **isSameAvg()** method:

```
// Determine if two averages are the same.  
// Notice the use of the wildcard.  
boolean isSameAvg(Stats<?> ob) {  
    if(average() == ob.average())  
        return true;  
  
    return false;  
}
```

Here, **Stats<?>** matches any **Stats** object, allowing any two **Stats** objects to have their averages compared. The following program demonstrates this:

```
// Use a wildcard.  
class Stats<T extends Number> {  
    T[] nums; // array of Number or subclass  
  
    // Pass the constructor a reference to  
    // an array of type Number or subclass.  
    Stats(T[] o) {  
        nums = o;  
    }  
  
    // Return type double in all cases.  
    double average() {  
        double sum = 0.0;  
  
        for(int i=0; i < nums.length; i++)  
            sum += nums[i].doubleValue();
```

```
        return sum / nums.length;
    }

    // Determine if two averages are the same.
    // Notice the use of the wildcard.
    boolean isSameAvg(Stats<?> ob) {
        if(average() == ob.average())
            return true;

        return false;
    }
}

// Demonstrate wildcard.
class WildcardDemo {
    public static void main(String[] args) {
        Integer[] inums = { 1, 2, 3, 4, 5 };
        Stats<Integer> iob = new Stats<Integer>(inums);
        double v = iob.average();
        System.out.println("iob average is " + v);

        Double[] dnums = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        Stats<Double> dob = new Stats<Double>(dnums);
        double w = dob.average();
        System.out.println("dob average is " + w);

        Float[] fnums = { 1.0F, 2.0F, 3.0F, 4.0F, 5.0F };
        Stats<Float> fob = new Stats<Float>(fnums);
        double x = fob.average();
        System.out.println("fob average is " + x);

        // See which arrays have same average.
        System.out.print("Averages of iob and dob ");
        if(iob.isSameAvg(dob))
            System.out.println("are the same.");
        else
            System.out.println("differ.");

        System.out.print("Averages of iob and fob ");
        if(iob.isSameAvg(fob))
            System.out.println("are the same.");
        else
            System.out.println("differ.");
    }
}
```

The output is shown here:

```
System.out.println("differ.");
}
} iob average is 3.0
dob average is 3.3
fob average is 3.0
Averages of iob and dob differ.
Averages of iob and fob are the same.
```

One last point: It is important to understand that the wildcard does not affect what type of **Stats** objects can be created. This is governed by the **extends** clause in the **Stats** declaration. The wildcard simply matches any *valid Stats* object.

Bounded Wildcards

Wildcard arguments can be bounded in much the same way that a type parameter can be bounded. A bounded wildcard is especially important when you are creating a generic type that will operate on a class hierarchy. To understand why, let's work through an example. Consider the following hierarchy of classes that encapsulate coordinates:

```
// Two-dimensional coordinates.  
class TwoD {  
    int x, y;  
  
    TwoD(int a, int b) {  
        x = a;  
        y = b;  
    }  
}  
  
// Three-dimensional coordinates.  
class ThreeD extends TwoD {  
    int z;  
  
    ThreeD(int a, int b, int c) {  
        super(a, b);  
        z = c;  
    }  
}  
  
// Four-dimensional coordinates.  
class FourD extends ThreeD {  
    int t;  
  
    FourD(int a, int b, int c, int d) {  
        super(a, b, c);  
        t = d;  
    }  
}
```

At the top of the hierarchy is **TwoD**, which encapsulates a two-dimensional, XY coordinate. **TwoD** is inherited by **ThreeD**, which adds a third dimension, creating an XYZ coordinate. **ThreeD** is inherited by **FourD**, which adds a fourth dimension (time), yielding a four-dimensional coordinate.

Shown next is a generic class called **Coords**, which stores an array of coordinates:

```
// This class holds an array of coordinate objects.
class Coords<T extends TwoD> {
    T[] coords;

    Coords(T[] o) { coords = o; }
}
```

Notice that **Coords** specifies a type parameter bounded by **TwoD**. This means that any array stored in a **Coords** object will contain objects of type **TwoD** or one of its subclasses.

Now, assume that you want to write a method that displays the X and Y coordinates for each element in the **coords** array of a **Coords** object. Because all types of **Coords** objects have at least two coordinates (X and Y), this is easy to do using a wildcard, as shown here:

```
static void showXY(Coords<?> c) {
    System.out.println("X Y Coordinates:");
    for(int i=0; i < c.coords.length; i++)
        System.out.println(c.coords[i].x + " " +
                           c.coords[i].y);
    System.out.println();
}
```

Because **Coords** is a bounded generic type that specifies **TwoD** as an upper bound, all objects that can be used to create a **Coords** object will be arrays of type **TwoD**, or of classes derived from **TwoD**. Thus, **showXY()** can display the contents of any **Coords** object.

However, what if you want to create a method that displays the X, Y, and Z coordinates of a **ThreeD** or **FourD** object? The trouble is that not all **Coords** objects will have three coordinates, because a **Coords<TwoD>** object will only have X and Y. Therefore, how do you write a method that displays the X, Y, and Z coordinates for **Coords<ThreeD>** and **Coords<FourD>** objects, while preventing that method from being used with **Coords<TwoD>** objects? The answer is the *bounded wildcard argument*.

A bounded wildcard specifies either an upper bound or a lower bound for the type argument. This enables you to restrict the types of objects upon which a method will operate. The most common bounded wildcard is the

upper bound, which is created using an **extends** clause in much the same way it is used to create a bounded type.

Using a bounded wildcard, it is easy to create a method that displays the X, Y, and Z coordinates of a **Coords** object, if that object actually has those three coordinates. For example, the following **showXYZ()** method shows the X, Y, and Z coordinates of the elements stored in a **Coords** object, if those elements are actually of type **ThreeD** (or are derived from **ThreeD**):

```
static void showXYZ(Coords<? extends ThreeD> c) {  
    System.out.println("X Y Z Coordinates:");  
    for(int i=0; i < c.coords.length; i++)  
        System.out.println(c.coords[i].x + " " +  
                           c.coords[i].y + " " +  
                           c.coords[i].z);  
    System.out.println();  
}
```

Notice that an **extends** clause has been added to the wildcard in the declaration of parameter **c**. It states that the **?** can match any type as long as it is **ThreeD**, or a class derived from **ThreeD**. Thus, the **extends** clause establishes an upper bound that the **?** can match. Because of this bound, **showXYZ()** can be called with references to objects of type **Coords<ThreeD>** or **Coords<FourD>**, but not with a reference of type **Coords<TwoD>**. Attempting to call **showXZY()** with a **Coords<TwoD>** reference results in a compile-time error, thus ensuring type safety.

Here is an entire program that demonstrates the actions of a bounded wildcard argument:

```
// Bounded Wildcard arguments.

// Two-dimensional coordinates.
class TwoD {
    int x, y;

    TwoD(int a, int b) {
        x = a;
        y = b;
    }
}

// Three-dimensional coordinates.
class ThreeD extends TwoD {
    int z;

    ThreeD(int a, int b, int c) {
        super(a, b);
        z = c;
    }
}

// Four-dimensional coordinates.
class FourD extends ThreeD {
    int t;

    FourD(int a, int b, int c, int d) {
        super(a, b, c);
        t = d;
    }
}

// This class holds an array of coordinate objects.
class Coords<T extends TwoD> {
    T[] coords;

    Coords(T[] o) { coords = o; }
}

// Demonstrate a bounded wildcard.
class BoundedWildcard {
    static void showXY(Coords<TwoD> o) {
```

```
        System.out.println(c.coords[i].x + " " +
                           c.coords[i].y + " " +
                           c.coords[i].z);
        System.out.println();
    }

static void showAll(Coords<? extends FourD> c) {
    System.out.println("X Y Z T Coordinates:");
    for(int i=0; i < c.coords.length; i++) {
        System.out.println(c.coords[i].x + " " +
                           c.coords[i].y + " " +
                           c.coords[i].z + " " +
                           c.coords[i].t);
    }
    System.out.println();
}

public static void main(String[] args) {
    TwoD[] td = {
        new TwoD(0, 0),
        new TwoD(7, 9),
        new TwoD(18, 4),
        new TwoD(-1, -23)
    };
    Coords<TwoD> tdlocs = new Coords<TwoD>(td);

    System.out.println("Contents of tdlocs.");
    showXY(tdlocs); // OK, is a TwoD
//    showXYZ(tdlocs); // Error, not a ThreeD
//    showAll(tdlocs); // Error, not a FourD

    // Now, create some FourD objects.
    FourD[] fd = {
        new FourD(1, 2, 3, 4),
        new FourD(6, 8, 14, 8),
        new FourD(22, 9, 4, 9),
        new FourD(3, -2, -23, 17)
    };
    Coords<FourD> fdlocs = new Coords<FourD>(fd);

    System.out.println("Contents of fdlocs.");
}
```

The output from the program is shown here:

```
// These are all OK.
showXY(fdlocs);
Contents of tdlocs.
X Y Coordinates:
0 0
}
} 7 9
18 4
-1 -23

Contents of fdlocs.
X Y Coordinates:
1 2
6 8
22 9
3 -2

X Y Z Coordinates:
1 2 3
6 8 14
22 9 4
3 -2 -23

X Y Z T Coordinates:
1 2 3 4
6 8 14 8
22 9 4 9
3 -2 -23 17
```

Notice these commented-out lines:

```
// showXYZ(tdlocs); // Error, not a ThreeD
// showAll(tdlocs); // Error, not a FourD
```

Because **tdlocs** is a **Coords(TwoD)** object, it cannot be used to call **showXYZ()** or **showAll()** because bounded wildcard arguments in their declarations prevent it. To prove this to yourself, try removing the comment symbols, and then attempt to compile the program. You will receive compilation errors because of the type mismatches.

In general, to establish an upper bound for a wildcard, use the following type of wildcard expression:

`<? extends superclass>`

where *superclass* is the name of the class that serves as the upper bound. Remember, this is an inclusive clause because the class forming the upper bound (that is, specified by *superclass*) is also within bounds.

You can also specify a lower bound for a wildcard by adding a **super** clause to a wildcard declaration. Here is its general form:

`<? super subclass>`

In this case, only classes that are superclasses of *subclass* are acceptable arguments. This is an inclusive clause.

Creating a Generic Method

As the preceding examples have shown, methods inside a generic class can make use of a class's type parameter and are, therefore, automatically generic relative to the type parameter. However, it is possible to declare a generic method that uses one or more type parameters of its own. Furthermore, it is possible to create a generic method that is enclosed within a non-generic class.

Let's begin with an example. The following program declares a non-generic class called **GenMethDemo** and a static generic method within that class called **isIn()**. The **isIn()** method determines if an object is a member of an array. It can be used with any type of object and array as long as the array contains objects that are compatible with the type of the object being sought.

```
// Demonstrate a simple generic method.
class GenMethDemo {

    // Determine if an object is in an array.
    static <T extends Comparable<T>, V extends T> boolean isIn(T x, V[] y) {
        for(int i=0; i < y.length; i++)
            if(x.equals(y[i])) return true;

        return false;
    }

    public static void main(String[] args) {

        // Use isIn() on Integers.
        Integer[] nums = { 1, 2, 3, 4, 5 };

        if(isIn(2, nums))
            System.out.println("2 is in nums");

        if(!isIn(7, nums))
            System.out.println("7 is not in nums");

        System.out.println();

        // Use isIn() on Strings.
        String[] strs = { "one", "two", "three",
                          "four", "five" };

        if(isIn("two", strs))
            System.out.println("two is in strs");

        if(!isIn("seven", strs))
            System.out.println("seven is not in strs");

        // Oops! Won't compile! Types must be compatible.
//        if(isIn("two", nums))
//            System.out.println("two is in strs");
    }
}
```

The output from the program is shown here:

```
2 is in nums  
7 is not in nums  
  
two is in strs  
seven is not in strs
```

Let's examine **isIn()** closely. First, notice how it is declared by this line:

```
static <T extends Comparable<T>, V extends T> boolean isIn(T x,  
V[] y) {
```

The type parameters are declared *before* the return type of the method. Also note that **T** extends **Comparable<T>**. **Comparable** is an interface declared in **java.lang**. A class that implements **Comparable** defines objects that can be ordered. Thus, requiring an upper bound of **Comparable** ensures that **isIn()** can be used only with objects that are capable of being compared. **Comparable** is generic, and its type parameter specifies the type of objects that it compares. (Shortly, you will see how to create a generic interface.) Next, notice that the type **V** is upper-bounded by **T**. Thus, **V** must either be the same as type **T**, or a subclass of **T**. This relationship enforces that **isIn()** can be called only with arguments that are compatible with each other. Also notice that **isIn()** is static, enabling it to be called independently of any object. Understand, though, that generic methods can be either static or non-static. There is no restriction in this regard.

Now, notice how **isIn()** is called within **main()** by use of the normal call syntax, without the need to specify type arguments. This is because the types of the arguments are automatically discerned, and the types of **T** and **V** are adjusted accordingly. For example, in the first call:

```
if(isIn(2, nums))
```

the type of the first argument is **Integer** (due to autoboxing), which causes **Integer** to be substituted for **T**. The base type of the second argument is also **Integer**, which makes **Integer** a substitute for **V**, too. In the second call, **String** types are used, and the types of **T** and **V** are replaced by **String**.

Although type inference will be sufficient for most generic method calls, you can explicitly specify the type argument if needed. For example, here is

how the first call to **isIn()** looks when the type arguments are specified:

```
GenMethDemo.<Integer, Integer>isIn(2, nums)
```

Of course, in this case, there is nothing gained by specifying the type arguments. Furthermore, JDK 8 improved type inference as it relates to methods. As a result, today there are fewer cases in which explicit type arguments are needed.

Now, notice the commented-out code, shown here:

```
//     if(isIn("two", nums))
//         System.out.println("two is in strs");
```

If you remove the comments and then try to compile the program, you will receive an error. The reason is that the type parameter **V** is bounded by **T** in the **extends** clause in **V**'s declaration. This means that **V** must be either type **T**, or a subclass of **T**. In this case, the first argument is of type **String**, making **T** into **String**, but the second argument is of type **Integer**, which is not a subclass of **String**. This causes a compile-time type-mismatch error. This ability to enforce type safety is one of the most important advantages of generic methods.

The syntax used to create **isIn()** can be generalized. Here is the syntax for a generic method:

```
<type-param-list> ret-type meth-name (param-list) { // ...
```

In all cases, *type-param-list* is a comma-separated list of type parameters. Notice that for a generic method, the type parameter list precedes the return type.

Generic Constructors

It is possible for constructors to be generic, even if their class is not. For example, consider the following short program:

```

// Use a generic constructor.
class GenCons {
    private double val;

    <T extends Number> GenCons(T arg) {
        val = arg.doubleValue();
    }

    void showVal() {
        System.out.println("val: " + val);
    }
}

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

        GenCons test = new GenCons(100);
        GenCons test2 = new GenCons(123.5F);

        test.showVal();
        test2.showVal();
    }
}

```

The output is shown here:

```

val: 100.0
val: 123.5

```

Because **GenCons()** specifies a parameter of a generic type, which must be a subclass of **Number**, **GenCons()** can be called with any numeric type, including **Integer**, **Float**, or **Double**. Therefore, even though **GenCons** is not a generic class, its constructor is generic.

Generic Interfaces

In addition to generic classes and methods, you can also have generic interfaces. Generic interfaces are specified just like generic classes. Here is an example. It creates an interface called **MinMax** that declares the

methods **min()** and **max()**, which are expected to return the minimum and maximum value of some set of objects.

```
// A generic interface example.

// A Min/Max interface.
interface MinMax<T extends Comparable<T>> {
    T min();
    T max();
}

// Now, implement MinMax
class MyClass<T extends Comparable<T>> implements MinMax<T> {
    T[] vals;

    MyClass(T[] o) { vals = o; }

    // Return the minimum value in vals.
    public T min() {
        T v = vals[0];

        for(int i=1; i < vals.length; i++)
            if(vals[i].compareTo(v) < 0) v = vals[i];

        return v;
    }

    // Return the maximum value in vals.
    public T max() {
        T v = vals[0];

        for(int i=1; i < vals.length; i++)
            if(vals[i].compareTo(v) > 0) v = vals[i];

        return v;
    }
}

class GenIFDemo {
    public static void main(String[] args) {
        Integer[] inums = {3, 6, 2, 8, 6};
        Character[] chs = {'b', 'r', 'p', 'w'};

        MyClass<Integer> iob = new MyClass<Integer>(inums);
        MyClass<Character> cob = new MyClass<Character>(chs).
```

```
        System.out.println("Max value in chs: " + cob.max());
        System.out.println("Min value in chs: " + cob.min());
    }
}
```

The output is shown here:

```
Max value in inums: 8
Min value in inums: 2
Max value in chs: w
Min value in chs: b
```

Although most aspects of this program should be easy to understand, a couple of key points need to be made. First, notice that **MinMax** is declared like this:

```
interface MinMax<T extends Comparable<T>> {
```

In general, a generic interface is declared in the same way as is a generic class. In this case, the type parameter is **T**, and its upper bound is **Comparable**. As explained earlier, **Comparable** is an interface defined by **java.lang** that specifies how objects are compared. Its type parameter specifies the type of the objects being compared.

Next, **MinMax** is implemented by **MyClass**. Notice the declaration of **MyClass**, shown here:

```
class MyClass<T extends Comparable<T>> implements MinMax<T> {
```

Pay special attention to the way that the type parameter **T** is declared by **MyClass** and then passed to **MinMax**. Because **MinMax** requires a type that implements **Comparable**, the implementing class (**MyClass** in this case) must specify the same bound. Furthermore, once this bound has been established, there is no need to specify it again in the **implements** clause. In fact, it would be wrong to do so. For example, this line is incorrect and won't compile:

```
// This is wrong!
class MyClass<T extends Comparable<T>>
    implements MinMax<T extends Comparable<T>> {
```

Once the type parameter has been established, it is simply passed to the interface without further modification.

In general, if a class implements a generic interface, then that class must also be generic, at least to the extent that it takes a type parameter that is passed to the interface. For example, the following attempt to declare **MyClass** is in error:

```
class MyClass implements MinMax<T> { // Wrong!
```

Because **MyClass** does not declare a type parameter, there is no way to pass one to **MinMax**. In this case, the identifier **T** is simply unknown, and the compiler reports an error. Of course, if a class implements a *specific type* of generic interface, such as shown here:

```
class MyClass implements MinMax<Integer> { // OK
```

then the implementing class does not need to be generic.

The generic interface offers two benefits. First, it can be implemented for different types of data. Second, it allows you to put constraints (that is, bounds) on the types of data for which the interface can be implemented. In the **MinMax** example, only types that implement the **Comparable** interface can be passed to **T**.

Here is the generalized syntax for a generic interface:

```
interface interface-name<type-param-list> { // ...
```

Here, *type-param-list* is a comma-separated list of type parameters. When a generic interface is implemented, you must specify the type arguments, as shown here:

```
class class-name<type-param-list>
    implements interface-name<type-arg-list> {
```

Raw Types and Legacy Code

Because support for generics did not exist prior to JDK 5, it was necessary to provide some transition path from old, pre-generics code. Furthermore, this transition path had to enable pre-generics code to remain functional

while at the same time being compatible with generics. In other words, pre-generics code had to be able to work with generics, and generic code had to be able to work with pre-generics code.

To handle the transition to generics, Java allows a generic class to be used without any type arguments. This creates a *raw type* for the class. This raw type is compatible with legacy code, which has no knowledge of generics. The main drawback to using the raw type is that the type safety of generics is lost.

Here is an example that shows a raw type in action:

```
// Demonstrate a raw type.
class Gen<T> {

    T ob; // declare an object of type T

    // Pass the constructor a reference to
    // an object of type T.
    Gen(T o) {
        ob = o;
    }

    // Return ob.
    T getOb() {
        return ob;
    }
}

// Demonstrate raw type.
class RawDemo {
    public static void main(String[] args) {

        // Create a Gen object for Integers.
        Gen<Integer> iOb = new Gen<Integer>(88);
```

```
// Create a Gen object for Strings.  
Gen<String> strOb = new Gen<String>("Generics Test");  
  
// Create a raw-type Gen object and give it  
// a Double value.  
Gen raw = new Gen(Double.valueOf(98.6));  
  
// Cast here is necessary because type is unknown.  
double d = (Double) raw.getOb();  
System.out.println("value: " + d);  
  
// The use of a raw type can lead to run-time  
// exceptions. Here are some examples.  
  
// // The following cast causes a run-time error!  
// int i = (Integer) raw.getOb(); // run-time error  
  
// This assignment overrides type safety.  
strOb = raw; // OK, but potentially wrong  
// String str = strOb.getOb(); // run-time error  
  
// This assignment also overrides type safety.  
raw = iOb; // OK, but potentially wrong  
// d = (Double) raw.getOb(); // run-time error  
}  
}
```

This program contains several interesting things. First, a raw type of the generic **Gen** class is created by the following declaration:

```
Gen raw = new Gen(Double.valueOf(98.6));
```

Notice that no type arguments are specified. In essence, this creates a **Gen** object whose type **T** is replaced by **Object**.

A raw type is not type safe. Thus, a variable of a raw type can be assigned a reference to any type of **Gen** object. The reverse is also allowed; a variable of a specific **Gen** type can be assigned a reference to a raw **Gen** object. However, both operations are potentially unsafe because the type checking mechanism of generics is circumvented.

This lack of type safety is illustrated by the commented-out lines at the end of the program. Let's examine each case. First, consider the following situation:

```
// int i = (Integer) raw.getOb(); // run-time error
```

In this statement, the value of **ob** inside **raw** is obtained, and this value is cast to **Integer**. The trouble is that **raw** contains a **Double** value, not an integer value. However, this cannot be detected at compile time because the type of **raw** is unknown. Thus, this statement fails at run time.

The next sequence assigns to a **strOb** (a reference of type **Gen<String>**) a reference to a raw **Gen** object:

```
strOb = raw; // OK, but potentially wrong
// String str = strOb.getOb(); // run-time error
```

The assignment itself is syntactically correct, but questionable. Because **strOb** is of type **Gen<String>**, it is assumed to contain a **String**. However, after the assignment, the object referred to by **strOb** contains a **Double**. Thus, at run time, when an attempt is made to assign the contents of **strOb** to **str**, a run-time error results because **strOb** now contains a **Double**. Thus, the assignment of a raw reference to a generic reference bypasses the type-safety mechanism.

The following sequence inverts the preceding case:

```
raw = iOb; // OK, but potentially wrong
// d = (Double) raw.getOb(); // run-time error
```

Here, a generic reference is assigned to a raw reference variable. Although this is syntactically correct, it can lead to problems, as illustrated by the second line. In this case, **raw** now refers to an object that contains an **Integer** object, but the cast assumes that it contains a **Double**. This error cannot be prevented at compile time. Rather, it causes a run-time error.

Because of the potential for danger inherent in raw types, **javac** displays *unchecked warnings* when a raw type is used in a way that might jeopardize type safety. In the preceding program, these lines generate unchecked warnings:

```
Gen raw = new Gen(Double.valueOf(98.6));  
  
strOb = raw; // OK, but potentially wrong
```

In the first line, it is the call to the **Gen** constructor without a type argument that causes the warning. In the second line, it is the assignment of a raw reference to a generic variable that generates the warning.

At first, you might think that this line should also generate an unchecked warning, but it does not:

```
raw = iOb; // OK, but potentially wrong
```

No compiler warning is issued because the assignment does not cause any *further* loss of type safety than had already occurred when **raw** was created.

One final point: You should limit the use of raw types to those cases in which you must mix legacy code with newer, generic code. Raw types are simply a transitional feature and not something that should be used for new code.

Generic Class Hierarchies

Generic classes can be part of a class hierarchy in just the same way as a non-generic class. Thus, a generic class can act as a superclass or be a subclass. The key difference between generic and non-generic hierarchies is that in a generic hierarchy, any type arguments needed by a generic superclass must be passed up the hierarchy by all subclasses. This is similar to the way that constructor arguments must be passed up a hierarchy.

Using a Generic Superclass

Here is a simple example of a hierarchy that uses a generic superclass:

```

// A simple generic class hierarchy.
class Gen<T> {
    T ob;

    Gen(T o) {
        ob = o;
    }

    // Return ob.
    T getOb() {
        return ob;
    }
}

// A subclass of Gen.
class Gen2<T> extends Gen<T> {
    Gen2(T o) {
        super(o);
    }
}

```

In this hierarchy, **Gen2** extends the generic class **Gen**. Notice how **Gen2** is declared by the following line:

```
class Gen2<T> extends Gen<T> {
```

The type parameter **T** is specified by **Gen2** and is also passed to **Gen** in the **extends** clause. This means that whatever type is passed to **Gen2** will also be passed to **Gen**. For example, this declaration,

```
Gen2<Integer> num = new Gen2<Integer>(100);
```

passes **Integer** as the type parameter to **Gen**. Thus, the **ob** inside the **Gen** portion of **Gen2** will be of type **Integer**.

Notice also that **Gen2** does not use the type parameter **T** except to support the **Gen** superclass. Thus, even if a subclass of a generic superclass would otherwise not need to be generic, it still must specify the type parameter(s) required by its generic superclass.

Of course, a subclass is free to add its own type parameters, if needed. For example, here is a variation on the preceding hierarchy in which **Gen2** adds a type parameter of its own:

```
// A subclass can add its own type parameters.
class Gen<T> {
    T ob; // declare an object of type T

    // Pass the constructor a reference to
    // an object of type T.
```

```

    Gen(T o) {
        ob = o;
    }

    // Return ob.
    T getOb() {
        return ob;
    }
}

// A subclass of Gen that defines a second
// type parameter, called V.
class Gen2<T, V> extends Gen<T> {
    V ob2;

    Gen2(T o, V o2) {
        super(o);
        ob2 = o2;
    }

    V getOb2() {
        return ob2;
    }
}

// Create an object of type Gen2.
class HierDemo {
    public static void main(String[] args) {

        // Create a Gen2 object for String and Integer.
        Gen2<String, Integer> x =
            new Gen2<String, Integer>("Value is: ", 99);

        System.out.print(x.getOb());
        System.out.println(x.getOb2());
    }
}

```

Notice the declaration of this version of **Gen2**, which is shown here:

```
class Gen2<T, V> extends Gen<T> {
```

Here, **T** is the type passed to **Gen**, and **V** is the type that is specific to **Gen2**. **V** is used to declare an object called **ob2**, and as a return type for the method **getOb2()**. In **main()**, a **Gen2** object is created in which type parameter **T** is **String**, and type parameter **V** is **Integer**. The program displays the following, expected, result:

```
Value is: 99
```

A Generic Subclass

It is perfectly acceptable for a non-generic class to be the superclass of a generic subclass. For example, consider this program:

```
// A non-generic class can be the superclass
// of a generic subclass.

// A non-generic class.
class NonGen {
    int num;

    NonGen(int i) {
        num = i;
    }

    int getnum() {
        return num;
    }
}

// A generic subclass.
class Gen<T> extends NonGen {
    T ob; // declare an object of type T

    // Pass the constructor a reference to
    // an object of type T.
    Gen(T o, int i) {
        super(i);
        ob = o;
    }

    // Return ob.
    T getOb() {
        return ob;
    }
}

// Create a Gen object.
class HierDemo2 {
    public static void main(String[] args) {

        // Create a Gen object for String.
        Gen<String> w = new Gen<String>("Hello", 47);

        System.out.print(w.getOb() + " ");
        System.out.println(w.getnum());
    }
}
```

The output from the program is shown here:

```
} Hello 47
```

In the program, notice how **Gen** inherits **NonGen** in the following declaration:

```
class Gen<T> extends NonGen {
```

Because **NonGen** is not generic, no type argument is specified. Thus, even though **Gen** declares the type parameter **T**, it is not needed by (nor can it be used by) **NonGen**. Thus, **NonGen** is inherited by **Gen** in the normal way. No special conditions apply.

Run-Time Type Comparisons Within a Generic Hierarchy

Recall the run-time type information operator **instanceof** that was introduced in [Chapter 13](#). As explained, **instanceof** determines if an object is an instance of a class. It returns true if an object is of the specified type or can be cast to the specified type. The **instanceof** operator can be applied to objects of generic classes. The following class demonstrates some of the type compatibility implications of a generic hierarchy:

```
// Use the instanceof operator with a generic class hierarchy.
class Gen<T> {
    T ob;

    Gen(T o) {
        ob = o;
    }

    // Return ob.
    T getOb() {
        return ob;
    }
}

// A subclass of Gen.
class Gen2<T> extends Gen<T> {
    Gen2(T o) {
        super(o);
    }
}

// Demonstrate run-time type ID implications of generic
// class hierarchy.
class HierDemo3 {
    public static void main(String[] args) {

        // Create a Gen object for Integers.
        Gen<Integer> iOb = new Gen<Integer>(88);
```

```
// Create a Gen2 object for Integers.  
Gen2<Integer> iOb2 = new Gen2<Integer>(99);  
  
// Create a Gen2 object for Strings.  
Gen2<String> strOb2 = new Gen2<String>("Generics Test");  
  
// See if iOb2 is some form of Gen2.  
if(iOb2 instanceof Gen2<?>)  
    System.out.println("iOb2 is instance of Gen2");  
  
// See if iOb2 is some form of Gen.  
if(iOb2 instanceof Gen<?>)  
    System.out.println("iOb2 is instance of Gen");  
  
System.out.println();  
  
// See if strOb2 is a Gen2.  
if(strOb2 instanceof Gen2<?>)  
    System.out.println("strOb2 is instance of Gen2");  
  
// See if strOb2 is a Gen.  
if(strOb2 instanceof Gen<?>)  
    System.out.println("strOb2 is instance of Gen");  
  
System.out.println();  
  
// See if iOb is an instance of Gen2, which it is not.  
if(iOb instanceof Gen2<?>)  
    System.out.println("iOb is instance of Gen2");  
  
// See if iOb is an instance of Gen, which it is.  
if(iOb instanceof Gen<?>)  
    System.out.println("iOb is instance of Gen");  
}  
}
```

The output from the program is shown here:

```
iOb2 is instance of Gen2
iOb2 is instance of Gen

strOb2 is instance of Gen2
strOb2 is instance of Gen

iOb is instance of Gen
```

In this program, **Gen2** is a subclass of **Gen**, which is generic on type parameter **T**. In **main()**, three objects are created. The first is **iOb**, which is an object of type **Gen<Integer>**. The second is **iOb2**, which is an instance of **Gen2<Integer>**. Finally, **strOb2** is an object of type **Gen2<String>**.

Then, the program performs these **instanceof** tests on the type of **iOb2**:

```
// See if iOb2 is some form of Gen2.
if(iOb2 instanceof Gen2<?>)
    System.out.println("iOb2 is instance of Gen2");

// See if iOb2 is some form of Gen.
if(iOb2 instanceof Gen<?>)
    System.out.println("iOb2 is instance of Gen");
```

As the output shows, both succeed. In the first test, **iOb2** is checked against **Gen2<?>**. This test succeeds because it simply confirms that **iOb2** is an object of some type of **Gen2** object. The use of the wildcard enables **instanceof** to determine if **iOb2** is an object of any type of **Gen2**. Next, **iOb2** is tested against **Gen<?>**, the superclass type. This is also true because **iOb2** is some form of **Gen**, the superclass. The next few lines in **main()** show the same sequence (and same results) for **strOb2**.

Next, **iOb**, which is an instance of **Gen<Integer>** (the superclass), is tested by these lines:

```
// See if iOb is an instance of Gen2, which it is not.
if(iOb instanceof Gen2<?>)
    System.out.println("iOb is instance of Gen2");

// See if iOb is an instance of Gen, which it is.
if(iOb instanceof Gen<?>)
    System.out.println("iOb is instance of Gen");
```

The first **if** fails because **iOb** is not some type of **Gen2** object. The second test succeeds because **iOb** is some type of **Gen** object.

Casting

You can cast one instance of a generic class into another only if the two are otherwise compatible and their type arguments are the same. For example, assuming the foregoing program, this cast is legal:

```
(Gen<Integer>) iOb2 // legal
```

because **iOb2** includes an instance of **Gen<Integer>**. But, this cast:

```
(Gen<Long>) iOb2 // illegal
```

is not legal because **iOb2** is not an instance of **Gen<Long>**.

Overriding Methods in a Generic Class

A method in a generic class can be overridden just like any other method. For example, consider this program in which the method **getOb()** is overridden:

```
// Overriding a generic method in a generic class.
class Gen<T> {
    T ob; // declare an object of type T
```

```
// Pass the constructor a reference to
// an object of type T.
Gen(T o) {
    ob = o;
}

// Return ob.
T getOb() {
    System.out.print("Gen's getOb(): " );
    return ob;
}
}

// A subclass of Gen that overrides getOb().
class Gen2<T> extends Gen<T> {

    Gen2(T o) {
        super(o);
    }

    // Override getOb().
    T getOb() {
        System.out.print("Gen2's getOb(): " );
        return ob;
    }
}

// Demonstrate generic method override.
class OverrideDemo {
    public static void main(String[] args) {

        // Create a Gen object for Integers.
        Gen<Integer> iOb = new Gen<Integer>(88);

        // Create a Gen2 object for Integers.
        Gen2<Integer> iOb2 = new Gen2<Integer>(99);

        // Create a Gen2 object for Strings.
        Gen2<String> strOb2 = new Gen2<String> ("Generics Test");

        System.out.println(iOb.getOb());
        System.out.println(iOb2.getOb());
    }
}
```

```
System.out.println(strOb2.getOb());
}
} Gen's getOb(): 88
} Gen2's getOb(): 99
Gen2's getOb(): Generics Test
```

As the output confirms, the overridden version of **getOb()** is called for objects of type **Gen2**, but the superclass version is called for objects of type **Gen**.

Type Inference with Generics

Beginning with JDK 7, it became possible to shorten the syntax used to create an instance of a generic type. To begin, consider the following generic class:

```
class MyClass<T, V> {
    T ob1;
    V ob2;

    MyClass(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }
    // ...
}
```

Prior to JDK 7, to create an instance of **MyClass**, you would have needed to use a statement similar to the following:

```
MyClass<Integer, String> mcOb =
    new MyClass<Integer, String>(98, "A String");
```

Here, the type arguments (which are **Integer** and **String**) are specified twice: first, when **mcOb** is declared, and second, when a **MyClass** instance is created via **new**. Since generics were introduced by JDK 5, this is the form required by all versions of Java prior to JDK 7. Although there is nothing wrong, per se, with this form, it is a bit more verbose than it needs to be. In the **new** clause, the type of the type arguments can be readily

inferred from the type of **mcOb**; therefore, there is really no reason that they need to be specified a second time. To address this situation, JDK 7 added a syntactic element that lets you avoid the second specification.

Today the preceding declaration can be rewritten as shown here:

```
MyClass<Integer, String> mcOb = new MyClass<>(98, "A String");
```

Notice that the instance creation portion simply uses \diamond , which is an empty type argument list. This is referred to as the *diamond* operator. It tells the compiler to infer the type arguments needed by the constructor in the **new** expression. The principal advantage of this type-inference syntax is that it shortens what are sometimes quite long declaration statements.

The preceding can be generalized. When type inference is used, the declaration syntax for a generic reference and instance creation has this general form:

$$\text{class-name} < \text{type-arg-list} > \text{var-name} = \text{new class-name} \diamond (\text{cons-arg-list});$$

Here, the type argument list of the constructor in the **new** clause is empty.

Type inference can also be applied to parameter passing. For example, if the following method is added to **MyClass**,

```
boolean isSame(MyClass<T, V> o) {
    if (ob1 == o.ob1 && ob2 == o.ob2) return true;
    else return false;
}
```

then the following call is legal:

```
if (mcOb.isSame(new MyClass<>(1, "test")))
    System.out.println("Same");
```

In this case, the type arguments for the argument passed to **isSame()** can be inferred from the parameter's type.

Most of the examples in this book will continue to use the full syntax when declaring instances of generic classes. This way, the examples will work with any Java compiler that supports generics. Using the full-length

syntax also makes it very clear precisely what is being created, which is important in sample code shown in a book. However, in your own code, the use of the type-inference syntax will streamline your declarations.

Local Variable Type Inference and Generics

As just explained, type inference is already supported for generics through the use of the diamond operator. However, you can also use the local variable type inference feature added by JDK 10 with a generic class. For example, assuming **MyClass** used in the preceding section, this declaration:

```
MyClass<Integer, String> mcOb =  
    new MyClass<Integer, String>(98, "A String");
```

can be rewritten like this using local variable type inference:

```
var mcOb = new MyClass<Integer, String>(98, "A String");
```

In this case, the type of **mcOb** is inferred to be **MyClass<Integer, String>** because that is the type of its initializer. Also notice that the use of **var** results in a shorter declaration than would be the case otherwise. In general, generic type names can often be quite long and (in some cases) complicated. The use of **var** is another way to substantially shorten such declarations. For the same reasons as just explained for the diamond operator, the remaining examples in this book will continue to use the full generic syntax, but in your own code the use of local variable type inference can be quite helpful.

Erasur

Usually, it is not necessary to know the details about how the Java compiler transforms your source code into object code. However, in the case of generics, some general understanding of the process is important because it explains why the generic features work as they do—and why their behavior is sometimes a bit surprising. For this reason, a brief discussion of how generics are implemented in Java is in order.

An important constraint that governed the way that generics were added to Java was the need for compatibility with previous versions of Java.

Simply put, generic code had to be compatible with preexisting, non-generic code. Thus, any changes to the syntax of the Java language, or to the JVM, had to avoid breaking older code. The way Java implements generics while satisfying this constraint is through the use of *erasure*.

In general, here is how erasure works: When your Java code is compiled, all generic type information is removed (erased). This means replacing type parameters with their bound type, which is **Object** if no explicit bound is specified, and then applying the appropriate casts (as determined by the type arguments) to maintain type compatibility with the types specified by the type arguments. The compiler also enforces this type compatibility. This approach to generics means that no type parameters exist at run time. They are simply a source-code mechanism.

Bridge Methods

Occasionally, the compiler will need to add a *bridge method* to a class to handle situations in which the type erasure of an overriding method in a subclass does not produce the same erasure as the method in the superclass. In this case, a method is generated that uses the type erasure of the superclass, and this method calls the method that has the type erasure specified by the subclass. Of course, bridge methods only occur at the bytecode level, are not seen by you, and are not available for your use.

Although bridge methods are not something that you will normally need to be concerned with, it is still instructive to see a situation in which one is generated. Consider the following program:

```
// A situation that creates a bridge method.
class Gen<T> {
    T ob; // declare an object of type T

    // Pass the constructor a reference to
    // an object of type T.
    Gen(T o) {
        ob = o;
    }

    // Return ob.
    T getOb() {
        return ob;
    }
}

// A subclass of Gen.
class Gen2 extends Gen<String> {

    Gen2(String o) {
        super(o);
    }

    // A String-specific override of getOb().
    String getOb() {
        System.out.print("You called String getOb(): ");
        return ob;
    }
}

// Demonstrate a situation that requires a bridge method.
class BridgeDemo {
    public static void main(String[] args) {

        // Create a Gen2 object for Strings.
        Gen2 strOb2 = new Gen2("Generics Test");

        System.out.println(strOb2.getOb());
    }
}
```

In the program, the subclass **Gen2** extends **Gen**, but does so using a **String**-specific version of **Gen**, as its declaration shows:

```
class Gen2 extends Gen<String> {
```

Furthermore, inside **Gen2**, **getOb()** is overridden with **String** specified as the return type:

```
// A String-specific override of getOb().  
String getOb() {  
    System.out.print("You called String getOb(): ");  
    return ob;  
}
```

All of this is perfectly acceptable. The only trouble is that because of type erasure, the expected form of **getOb()** will be

```
Object getOb() { // ...
```

To handle this problem, the compiler generates a bridge method with the preceding signature that calls the **String** version. Thus, if you examine the class file for **Gen2** by using **javap**, you will see the following methods:

```
class Gen2 extends Gen<java.lang.String> {  
    Gen2(java.lang.String);  
    java.lang.String getOb();  
    java.lang.Object getOb(); // bridge method  
}
```

As you can see, the bridge method has been included. (The comment was added by the author and not by **javap**, and the precise output you see may vary based on the version of Java that you are using.)

There is one last point to make about this example. Notice that the only difference between the two **getOb()** methods is their return type. Normally, this would cause an error, but because this does not occur in your source code, it does not cause a problem and is handled correctly by the JVM.

Ambiguity Errors

The inclusion of generics gives rise to another type of error that you must guard against: *ambiguity*. Ambiguity errors occur when erasure causes two seemingly distinct generic declarations to resolve to the same erased type, causing a conflict. Here is an example that involves method overloading:

```
// Ambiguity caused by erasure on
// overloaded methods.
class MyGenClass<T, V> {
    T ob1;
    V ob2;

    // ...

    // These two overloaded methods are ambiguous
    // and will not compile.
    void set(T o) {
        ob1 = o;
    }

    void set(V o) {
        ob2 = o;
    }
}
```

Notice that **MyGenClass** declares two generic types: **T** and **V**. Inside **MyGenClass**, an attempt is made to overload **set()** based on parameters of type **T** and **V**. This looks reasonable because **T** and **V** appear to be different types. However, there are two ambiguity problems here.

First, as **MyGenClass** is written, there is no requirement that **T** and **V** actually be different types. For example, it is perfectly correct (in principle) to construct a **MyGenClass** object as shown here:

```
MyGenClass<String, String> obj = new MyGenClass<String, String>()
```

In this case, both **T** and **V** will be replaced by **String**. This makes both versions of **set()** identical, which is, of course, an error.

The second and more fundamental problem is that the type erasure of **set()** reduces both versions to the following:

```
void set(Object o) { // ...
```

Thus, the overloading of **set()** as attempted in **MyGenClass** is inherently ambiguous.

Ambiguity errors can be tricky to fix. For example, if you know that **V** will always be some type of **Number**, you might try to fix **MyGenClass** by rewriting its declaration as shown here:

```
class MyGenClass<T, V extends Number> { // almost OK!
```

This change causes **MyGenClass** to compile, and you can even instantiate objects like the one shown here:

```
MyGenClass<String, Number> x = new MyGenClass<String, Number>()
```

This works because Java can accurately determine which method to call. However, ambiguity returns when you try this line:

```
MyGenClass<Number, Number> x = new MyGenClass<Number, Number>()
```

In this case, since both **T** and **V** are **Number**, which version of **set()** is to be called? The call to **set()** is now ambiguous.

Frankly, in the preceding example, it would be much better to use two separate method names, rather than trying to overload **set()**. Often, the solution to ambiguity involves the restructuring of the code, because ambiguity frequently means that you have a conceptual error in your design.

Some Generic Restrictions

There are a few restrictions that you need to keep in mind when using generics. They involve creating objects of a type parameter, static members, exceptions, and arrays. Each is examined here.

Type Parameters Can't Be Instantiated

It is not possible to create an instance of a type parameter. For example, consider this class:

```
// Can't create an instance of T.
class Gen<T> {
    T ob;

    Gen() {
        ob = new T(); // Illegal!!!
    }
}
```

Here, it is illegal to attempt to create an instance of **T**. The reason should be easy to understand: the compiler does not know what type of object to create. **T** is simply a placeholder.

Restrictions on Static Members

No **static** member can use a type parameter declared by the enclosing class. For example, both of the **static** members of this class are illegal:

```
class Wrong<T> {
    // Wrong, no static variables of type T.
    static T ob;

    // Wrong, no static method can use T.
    static T getOb() {
        return ob;
    }
}
```

Although you can't declare **static** members that use a type parameter declared by the enclosing class, you can declare **static** generic methods, which define their own type parameters, as was done earlier in this chapter.

Generic Array Restrictions

There are two important generics restrictions that apply to arrays. First, you cannot instantiate an array whose element type is a type parameter. Second,

you cannot create an array of type-specific generic references. The following short program shows both situations:

```
// Generics and arrays.
class Gen<T extends Number> {
    T ob;

    T[] vals; // OK

    Gen(T o, T[] nums) {
        ob = o;

        // This statement is illegal.
        // vals = new T[10]; // can't create an array of T

        // But, this statement is OK.
        vals = nums; // OK to assign reference to existent array
    }
}

class GenArrays {
    public static void main(String[] args) {
        Integer[] n = { 1, 2, 3, 4, 5 };

        Gen<Integer> iOb = new Gen<Integer>(50, n);

        // Can't create an array of type-specific generic references.
        // Gen<Integer>[] gens = new Gen<Integer>[10]; // Wrong!

        // This is OK.
        Gen<?>[] gens = new Gen<?>[10]; // OK
    }
}
```

As the program shows, it's valid to declare a reference to an array of type **T**, as this line does:

```
T[] vals; // OK
```

But, you cannot instantiate an array of **T**, as this commented-out line attempts:

```
// vals = new T[10]; // can't create an array of T
```

The reason you can't create an array of **T** is that there is no way for the compiler to know what type of array to actually create.

However, you can pass a reference to a type-compatible array to **Gen()** when an object is created and assign that reference to **vals**, as the program does in this line:

```
vals = nums; // OK to assign reference to existent array
```

This works because the array passed to **Gen** has a known type, which will be the same type as **T** at the time of object creation.

Inside **main()**, notice that you can't declare an array of references to a specific generic type. That is, this line

```
// Gen<Integer>[] gens = new Gen<Integer>[10]; // Wrong!
```

won't compile.

You *can* create an array of references to a generic type if you use a wildcard, however, as shown here:

```
Gen<?>[] gens = new Gen<?>[10]; // OK
```

This approach is better than using an array of raw types, because at least some type checking will still be enforced.

Generic Exception Restriction

A generic class cannot extend **Throwable**. This means that you cannot create generic exception classes.

CHAPTER

15

Lambda Expressions

During Java’s ongoing development and evolution, many features have been added since its original 1.0 release. However, two stand out because they have profoundly reshaped the language, fundamentally changing the way that code is written. The first was the addition of generics, added by JDK 5. (See [Chapter 14](#).) The second is the *lambda expression*, which is the subject of this chapter.

Added by JDK 8, lambda expressions (and their related features) significantly enhanced Java because of two primary reasons. First, they added new syntax elements that increased the expressive power of the language. In the process, they streamlined the way that certain common constructs are implemented. Second, the addition of lambda expressions resulted in new capabilities being incorporated into the API library. Among these new capabilities are the ability to more easily take advantage of the parallel processing capabilities of multicore environments, especially as it relates to the handling of for-each style operations, and the new stream API, which supports pipeline operations on data. The addition of lambda expressions also provided the catalyst for other new Java features, including the default method (described in [Chapter 9](#)), which lets you define default behavior for an interface method, and the method reference (described here), which lets you refer to a method without executing it.

In the final analysis, in much the same way that generics reshaped Java several years ago, lambda expressions continue to reshape Java today. Simply put, lambda expressions will impact virtually all Java programmers. They truly are that important.

Introducing Lambda Expressions

Key to understanding Java's implementation of lambda expressions are two constructs. The first is the lambda expression itself. The second is the functional interface. Let's begin with a simple definition of each.

A *lambda expression* is, essentially, an anonymous (that is, unnamed) method. However, this method is not executed on its own. Instead, it is used to implement a method defined by a functional interface. Thus, a lambda expression results in a form of anonymous class. Lambda expressions are also commonly referred to as *closures*.

A *functional interface* is an interface that contains one and only one abstract method. Normally, this method specifies the intended purpose of the interface. Thus, a functional interface typically represents a single action. For example, the standard interface **Runnable** is a functional interface because it defines only one method: **run()**. Therefore, **run()** defines the action of **Runnable**. Furthermore, a functional interface defines the *target type* of a lambda expression. Here is a key point: a lambda expression can be used only in a context in which its target type is specified. One other thing: a functional interface is sometimes referred to as a *SAM type*, where SAM stands for Single Abstract Method.

NOTE A functional interface may specify any public method defined by **Object**, such as **equals()**, without affecting its "functional interface" status. The public **Object** methods are considered implicit members of a functional interface because they are automatically implemented by an instance of a functional interface.

Let's now look more closely at both lambda expressions and functional interfaces.

Lambda Expression Fundamentals

The lambda expression introduced a new syntax element and operator into the Java language. The new operator, sometimes referred to as the *lambda operator* or the *arrow operator*, is \rightarrow . It divides a lambda expression into two parts. The left side specifies any parameters required by the lambda expression. (If no parameters are needed, an empty parameter list is used.) On the right side is the *lambda body*, which specifies the actions of the lambda expression. The \rightarrow can be verbalized as "becomes" or "goes to."

Java defines two types of lambda bodies. One consists of a single expression, and the other type consists of a block of code. We will begin with lambdas that define a single expression. Lambdas with block bodies are discussed later in this chapter.

At this point, it will be helpful to look at a few examples of lambda expressions before continuing. Let's begin with what is probably the simplest type of lambda expression you can write. It evaluates to a constant value and is shown here:

```
() -> 123.45
```

This lambda expression takes no parameters; thus, the parameter list is empty. It returns the constant value 123.45. Therefore, it is similar to the following method:

```
double myMeth() { return 123.45; }
```

Of course, the method defined by a lambda expression does not have a name.

A slightly more interesting lambda expression is shown here:

```
() -> Math.random() * 100
```

This lambda expression obtains a pseudo-random value from **Math.random()**, multiplies it by 100, and returns the result. It, too, does not require a parameter.

When a lambda expression requires a parameter, it is specified in the parameter list on the left side of the lambda operator. Here is a simple example:

```
(n) -> (n % 2) == 0
```

This lambda expression returns **true** if the value of parameter **n** is even. Although it is possible to explicitly specify the type of a parameter, such as **n** in this case, often you won't need to do so because in many cases its type can be inferred. Like a named method, a lambda expression can specify as many parameters as needed.

Functional Interfaces

As stated, a functional interface is an interface that specifies only one abstract method. If you have been programming in Java for some time, you might at first think that all interface methods are implicitly abstract.

Although this was true prior to JDK 8, the situation has changed. As explained in [Chapter 9](#), beginning with JDK 8, it is possible to specify a default implementation for a method declared in an interface. Private and static interface methods also supply an implementation. As a result, today, an interface method is abstract only if it does not specify an implementation. Because non-default, non-static, non-private interface methods are implicitly abstract, there is no need to use the **abstract** modifier (although you can specify it, if you like).

Here is an example of a functional interface:

```
interface MyNumber {  
    double getValue();  
}
```

In this case, the method **getValue()** is implicitly abstract, and it is the only method defined by **MyNumber**. Thus, **MyNumber** is a functional interface, and its function is defined by **getValue()**.

As mentioned earlier, a lambda expression is not executed on its own. Rather, it forms the implementation of the abstract method defined by the functional interface that specifies its target type. As a result, a lambda expression can be specified only in a context in which a target type is defined. One of these contexts is created when a lambda expression is assigned to a functional interface reference. Other target type contexts include variable initialization, **return** statements, and method arguments, to name a few.

Let's work through an example that shows how a lambda expression can be used in an assignment context. First, a reference to the functional interface **MyNumber** is declared:

```
// Create a reference to a MyNumber instance.  
MyNumber myNum;
```

Next, a lambda expression is assigned to that interface reference:

```
// Use a lambda in an assignment context.  
myNum = () -> 123.45;
```

When a lambda expression occurs in a target type context, an instance of a class is automatically created that implements the functional interface, with the lambda expression defining the behavior of the abstract method declared by the functional interface. When that method is called through the target, the lambda expression is executed. Thus, a lambda expression gives us a way to transform a code segment into an object.

In the preceding example, the lambda expression becomes the implementation for the `getValue()` method. As a result, the following displays the value 123.45:

```
// Call getValue(), which is implemented by the previously assigned  
// lambda expression.  
System.out.println(myNum.getValue());
```

Because the lambda expression assigned to `myNum` returns the value 123.45, that is the value obtained when `getValue()` is called.

In order for a lambda expression to be used in a target type context, the type of the abstract method and the type of the lambda expression must be compatible. For example, if the abstract method specifies two `int` parameters, then the lambda must specify two parameters whose type either is explicitly `int` or can be implicitly inferred as `int` by the context. In general, the type and number of the lambda expression's parameters must be compatible with the method's parameters; the return types must be compatible; and any exceptions thrown by the lambda expression must be acceptable to the method.

Some Lambda Expression Examples

With the preceding discussion in mind, let's look at some simple examples that illustrate the basic lambda expression concepts. The first example puts together the pieces shown in the foregoing section.

```
// Demonstrate a simple lambda expression.

// A functional interface.
interface MyNumber {
    double getValue();
}

class LambdaDemo {
    public static void main(String[] args)
    {
        MyNumber myNum; // declare an interface reference

        // Here, the lambda expression is simply a constant expression.
        // When it is assigned to myNum, a class instance is
        // constructed in which the lambda expression implements
        // the getValue() method in MyNumber.
        myNum = () -> 123.45;

        // Call getValue(), which is provided by the previously assigned
        // lambda expression.
        System.out.println("A fixed value: " + myNum.getValue());

        // Here, a more complex expression is used.
        myNum = () -> Math.random() * 100;

        // These call the lambda expression in the previous line.
        System.out.println("A random value: " + myNum.getValue());
        System.out.println("Another random value: " + myNum.getValue());

        // A lambda expression must be compatible with the method
        // defined by the functional interface. Therefore, this won't work:
        // myNum = () -> "123.03"; // Error!
    }
}
```

Sample output from the program is shown here:

```
A fixed value: 123.45
A random value: 88.90663650412304
Another random value: 53.00582701784129
```

As mentioned, the lambda expression must be compatible with the abstract method that it is intended to implement. For this reason, the commented-out line at the end of the preceding program is illegal because a value of type **String** is not compatible with **double**, which is the return type required by **getValue()**.

The next example shows the use of a parameter with a lambda expression:

```
// Demonstrate a lambda expression that takes a parameter.

// Another functional interface.
interface NumericTest {
    boolean test(int n);
}

class LambdaDemo2 {
    public static void main(String[] args)
    {
        // A lambda expression that tests if a number is even.
        NumericTest isEven = (n) -> (n % 2)==0;

        if(isEven.test(10)) System.out.println("10 is even");
        if(!isEven.test(9)) System.out.println("9 is not even");

        // Now, use a lambda expression that tests if a number
        // is non-negative.
        NumericTest isNonNeg = (n) -> n >= 0;

        if(isNonNeg.test(1)) System.out.println("1 is non-negative");
        if(!isNonNeg.test(-1)) System.out.println("-1 is negative");
    }
}
```

The output from this program is shown here:

```
10 is even
9 is not even
1 is non-negative
-1 is negative
```

This program demonstrates a key fact about lambda expressions that warrants close examination. Pay special attention to the lambda expression that performs the test for evenness. It is shown again here:

```
(n) -> (n % 2)==0
```

Notice that the type of **n** is not specified. Rather, its type is inferred from the context. In this case, its type is inferred from the parameter type of **test()** as defined by the **NumericTest** interface, which is **int**. It is also possible to explicitly specify the type of a parameter in a lambda expression. For example, this is also a valid way to write the preceding:

```
(int n) -> (n % 2)==0
```

Here, **n** is explicitly specified as **int**. Usually it is not necessary to explicitly specify the type, but you can in those situations that require it. Beginning with JDK 11, you can also use **var** to explicitly indicate local variable type inference for a lambda expression parameter.

This program demonstrates another important point about lambda expressions: A functional interface reference can be used to execute any lambda expression that is compatible with it. Notice that the program defines two different lambda expressions that are compatible with the **test()** method of the functional interface **NumericTest**. The first, called **isEven**, determines if a value is even. The second, called **isNonNeg**, checks if a value is non-negative. In each case, the value of the parameter **n** is tested. Because each lambda expression is compatible with **test()**, each can be executed through a **NumericTest** reference.

One other point before moving on: When a lambda expression has only one parameter, it is not necessary to surround the parameter name with parentheses when it is specified on the left side of the lambda operator. For example, this is also a valid way to write the lambda expression used in the program:

```
n -> (n % 2)==0
```

For consistency, this book will surround all lambda expression parameter lists with parentheses, even those containing only one parameter. Of course, you are free to adopt a different style.

The next program demonstrates a lambda expression that takes two parameters. In this case, the lambda expression tests if one number is a factor of another.

```
// Demonstrate a lambda expression that takes two parameters.

interface NumericTest2 {
    boolean test(int n, int d);
}

class LambdaDemo3 {
    public static void main(String[] args)
    {
        // This lambda expression determines if one number is
        // a factor of another.
        NumericTest2 isFactor = (n, d) -> (n % d) == 0;

        if(isFactor.test(10, 2))
            System.out.println("2 is a factor of 10");

        if(!isFactor.test(10, 3))
            System.out.println("3 is not a factor of 10");
    }
}
```

The output is shown here:

```
2 is a factor of 10
3 is not a factor of 10
```

In this program, the functional interface **NumericTest2** defines the **test()** method:

```
boolean test(int n, int d);
```

In this version, **test()** specifies two parameters. Thus, for a lambda expression to be compatible with **test()**, the lambda expression must also specify two parameters. Notice how they are specified:

```
(n, d) -> (n % d) == 0
```

The two parameters, **n** and **d**, are specified in the parameter list, separated by commas. This example can be generalized. Whenever more than one parameter is required, the parameters are specified, separated by commas, in a parenthesized list on the left side of the lambda operator.

Here is an important point about multiple parameters in a lambda expression: If you need to explicitly declare the type of a parameter, then all of the parameters must have declared types. For example, this is legal:

```
(int n, int d) -> (n % d) == 0
```

But this is not:

```
(int n, d) -> (n % d) == 0
```

Block Lambda Expressions

The body of the lambdas shown in the preceding examples consist of a single expression. These types of lambda bodies are referred to as *expression bodies*, and lambdas that have expression bodies are sometimes called *expression lambdas*. In an expression body, the code on the right side of the lambda operator must consist of a single expression. While expression lambdas are quite useful, sometimes the situation will require more than a single expression. To handle such cases, Java supports a second type of lambda expression in which the code on the right side of the lambda operator consists of a block of code that can contain more than one statement. This type of lambda body is called a *block body*. Lambdas that have block bodies are sometimes referred to as *block lambdas*.

A block lambda expands the types of operations that can be handled within a lambda expression because it allows the body of the lambda to contain multiple statements. For example, in a block lambda you can declare variables, use loops, specify **if** and **switch** statements, create nested blocks, and so on. A block lambda is easy to create. Simply enclose the body within braces as you would any other block of statements.

Aside from allowing multiple statements, block lambdas are used much like the expression lambdas just discussed. One key difference, however, is that you must explicitly use a **return** statement to return a value. This is

necessary because a block lambda body does not represent a single expression.

Here is an example that uses a block lambda to compute and return the factorial of an **int** value:

```
// A block lambda that computes the factorial of an int value.

interface NumericFunc {
    int func(int n);
}

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

        // This block lambda computes the factorial of an int value.
        NumericFunc factorial = (n) -> {
            int result = 1;

            for(int i=1; i <= n; i++)
                result = i * result;

            return result;
        };

        System.out.println("The factorial of 3 is " + factorial.func(3));
        System.out.println("The factorial of 5 is " + factorial.func(5));
    }
}
```

The output is shown here:

```
The factorial of 3 is 6
The factorial of 5 is 120
```

In the program, notice that the block lambda declares a variable called **result**, uses a **for** loop, and has a **return** statement. These are legal inside a block lambda body. In essence, the block body of a lambda is similar to a method body. One other point: When a **return** statement occurs within a

lambda expression, it simply causes a return from the lambda. It does not cause an enclosing method to return.

Another example of a block lambda is shown in the following program. It reverses the characters in a string.

```
// A block lambda that reverses the characters in a string.

interface StringFunc {
    String func(String n);
}

class BlockLambdaDemo2 {
    public static void main(String[] args)
    {

        // This block lambda reverses the characters in a string.
        StringFunc reverse = (str) ->  {
            String result = "";
            int i;

            for(i = str.length()-1; i >= 0; i--)
                result += str.charAt(i);

            return result;
        };

        System.out.println("Lambda reversed is " +
                           reverse.func("Lambda"));
        System.out.println("Expression reversed is " +
                           reverse.func("Expression"));
    }
}
```

The output is shown here:

```
Lambda reversed is adbmaL
Expression reversed is noisserpxE
```

In this example, the functional interface **StringFunc** declares the **func()** method. This method takes a parameter of type **String** and has a return type

of **String**. Thus, in the **reverse** lambda expression, the type of **str** is inferred to be **String**. Notice that the **charAt()** method is called on **str**. This is legal because of the inference that **str** is of type **String**.

Generic Functional Interfaces

A lambda expression itself cannot specify type parameters. Thus, a lambda expression cannot be generic. (Of course, because of type inference, all lambda expressions exhibit some “generic-like” qualities.) However, the functional interface associated with a lambda expression can be generic. In this case, the target type of the lambda expression is determined, in part, by the type argument or arguments specified when a functional interface reference is declared.

To understand the value of generic functional interfaces, consider this: The two examples in the previous section used two different functional interfaces, one called **NumericFunc** and the other called **StringFunc**. However, both defined a method called **func()** that took one parameter and returned a result. In the first case, the type of the parameter and return type was **int**. In the second case, the parameter and return type was **String**. Thus, the only difference between the two methods was the type of data they required. Instead of having two functional interfaces whose methods differ only in their data types, it is possible to declare one generic interface that can be used to handle both circumstances. The following program shows this approach:

```
// Use a generic functional interface with lambda expressions.

// A generic functional interface.
interface SomeFunc<T> {
    T func(T t);
}

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

        // Use a String-based version of SomeFunc.
        SomeFunc<String> reverse = (str) -> {
            String result = "";
            int i;

            for(i = str.length()-1; i >= 0; i--)
                result += str.charAt(i);

            return result;
        };

        System.out.println("Lambda reversed is " +
                           reverse.func("Lambda"));
        System.out.println("Expression reversed is " +
                           reverse.func("Expression"));

        // Now, use an Integer-based version of SomeFunc.
        SomeFunc<Integer> factorial = (n) -> {
            int result = 1;

            for(int i=1; i <= n; i++)
                result = i * result;

            return result;
        };

        System.out.println("The factorial of 3 is " + factorial.func(3));
        System.out.println("The factorial of 5 is " + factorial.func(5));
    }
}
```

The output is shown here:

```
Lambda reversed is adbmaL
Expression reversed is noisserpxE
The factorial of 3 is 6
The factorial of 5 is 120
```

In the program, the generic functional interface **SomeFunc** is declared as shown here:

```
interface SomeFunc<T> {
    T func(T t);
}
```

Here, **T** specifies both the return type and the parameter type of **func()**. This means that it is compatible with any lambda expression that takes one parameter and returns a value of the same type.

The **SomeFunc** interface is used to provide a reference to two different types of lambdas. The first uses type **String**. The second uses type **Integer**. Thus, the same functional interface can be used to refer to the **reverse** lambda and the **factorial** lambda. Only the type argument passed to **SomeFunc** differs.

Passing Lambda Expressions as Arguments

As explained earlier, a lambda expression can be used in any context that provides a target type. One of these is when a lambda expression is passed as an argument. In fact, passing a lambda expression as an argument is a common use of lambdas. Moreover, it is a very powerful use because it gives you a way to pass executable code as an argument to a method. This greatly enhances the expressive power of Java.

To pass a lambda expression as an argument, the type of the parameter receiving the lambda expression argument must be of a functional interface type compatible with the lambda. Although using a lambda expression as an argument is straightforward, it is still helpful to see it in action. The following program demonstrates the process:

```
// Use lambda expressions as an argument to a method.

interface StringFunc {
    String func(String n);
}

class LambdasAsArgumentsDemo {

    // This method has a functional interface as the type of
    // its first parameter. Thus, it can be passed a reference to
    // any instance of that interface, including the instance created
    // by a lambda expression.
    // The second parameter specifies the string to operate on.
    static String stringOp(StringFunc sf, String s) {
        return sf.func(s);
    }
}
```

```
public static void main(String[] args)
{
    String inStr = "Lambdas add power to Java";
    String outStr;

    System.out.println("Here is input string: " + inStr);

    // Here, a simple expression lambda that uppercases a string
    // is passed to stringOp( ).
    outStr = stringOp((str) -> str.toUpperCase(), inStr);
    System.out.println("The string in uppercase: " + outStr);

    // This passes a block lambda that removes spaces.
    outStr = stringOp((str) -> {
        String result = "";
        int i;

        for(i = 0; i < str.length(); i++)
        if(str.charAt(i) != ' ')
            result += str.charAt(i);

        return result;
    }, inStr);

    System.out.println("The string with spaces removed: " + outStr);

    // Of course, it is also possible to pass a StringFunc instance
    // created by an earlier lambda expression. For example,
    // after this declaration executes, reverse refers to an
    // instance of StringFunc.
    StringFunc reverse = (str) -> {
        String result = "";
        int i;

        for(i = str.length()-1; i >= 0; i--)
            result += str.charAt(i);

        return result;
    };

    // Now reverse can be passed as the first parameter to stringOp()
}
```

The output is shown here:

```
// Since it refers to a StringFunc object.  
System.out.println("The string reversed: " +  
Here is input string: Lambdas add power to Java  
The string in uppercase: LAMBDAS ADD POWER TO JAVA  
The string with spaces removed: LambdasaddpowertoJava  
The string reversed: avaJ ot rewop dda sadbmaL
```

In the program, first notice the **stringOp()** method. It has two parameters. The first is of type **StringFunc**, which is a functional interface. Thus, this parameter can receive a reference to any instance of **StringFunc**, including one created by a lambda expression. The second argument of **stringOp()** is of type **String**, and this is the string operated on.

Next, notice the first call to **stringOp()**, shown again here:

```
outStr = stringOp((str) -> str.toUpperCase(), inStr);
```

Here, a simple expression lambda is passed as an argument. When this occurs, an instance of the functional interface **StringFunc** is created and a reference to that object is passed to the first parameter of **stringOp()**. Thus, the lambda code, embedded in a class instance, is passed to the method. The target type context is determined by the type of parameter. Because the lambda expression is compatible with that type, the call is valid. Embedding simple lambdas, such as the one just shown, inside a method call is often a convenient technique—especially when the lambda expression is intended for a single use.

Next, the program passes a block lambda to **stringOp()**. This lambda removes spaces from a string. It is shown again here:

```
outStr = stringOp((str) -> {  
    String result = "";  
    int i;  
  
    for(i = 0; i < str.length(); i++)  
        if(str.charAt(i) != ' ')  
            result += str.charAt(i);  
  
    return result;  
}, inStr);
```

Although this uses a block lambda, the process of passing the lambda expression is the same as just described for the simple expression lambda. In this case, however, some programmers will find the syntax a bit awkward.

When a block lambda seems overly long to embed in a method call, it is an easy matter to assign that lambda to a functional interface variable, as the previous examples have done. Then, you can simply pass that reference to the method. This technique is shown at the end of the program. There, a block lambda is defined that reverses a string. This lambda is assigned to **reverse**, which is a reference to a **StringFunc** instance. Thus, **reverse** can be used as an argument to the first parameter of **stringOp()**. The program then calls **stringOp()**, passing in **reverse** and the string on which to operate. Because the instance obtained by the evaluation of each lambda expression is an implementation of **StringFunc**, each can be used as the first parameter to **stringOp()**.

One last point: In addition to variable initialization, assignment, and argument passing, the following also constitute target type contexts: casts, the **?** operator, array initializers, **return** statements, and lambda expressions themselves.

Lambda Expressions and Exceptions

A lambda expression can throw an exception. However, if it throws a checked exception, then that exception must be compatible with the exception(s) listed in the **throws** clause of the abstract method in the functional interface. Here is an example that illustrates this fact. It computes the average of an array of **double** values. If a zero-length array is passed, however, it throws the custom exception **EmptyArrayException**. As the example shows, this exception is listed in the **throws** clause of **func()** declared inside the **DoubleNumericArrayFunc** functional interface.

```

// Throw an exception from a lambda expression.

interface DoubleNumericArrayFunc {
    double func(double[] n) throws EmptyArrayException;
}

class EmptyArrayException extends Exception {
    EmptyArrayException() {
        super("Array Empty");
    }
}

class LambdaExceptionDemo {

    public static void main(String[] args) throws EmptyArrayException
    {
        double[] values = { 1.0, 2.0, 3.0, 4.0 };

        // This block lambda computes the average of an array of doubles.
        DoubleNumericArrayFunc average = (n) -> {
            double sum = 0;

            if(n.length == 0)
                throw new EmptyArrayException();

            for(int i=0; i < n.length; i++)
                sum += n[i];

            return sum / n.length;
        };

        System.out.println("The average is " + average.func(values));

        // This causes an exception to be thrown.
        System.out.println("The average is " + average.func(new double[0]));
    }
}

```

The first call to **average.func()** returns the value 2.5. The second call, which passes a zero-length array, causes an **EmptyArrayException** to be thrown. Remember, the inclusion of the **throws** clause in **func()** is

necessary. Without it, the program will not compile because the lambda expression will no longer be compatible with `func()`.

This example demonstrates another important point about lambda expressions. Notice that the parameter specified by `func()` in the functional interface `DoubleNumericArrayFunc` is an array. However, the parameter to the lambda expression is simply `n`, rather than `n[]`. Remember, the type of a lambda expression parameter will be inferred from the target context. In this case, the target context is `double[]`, so the type of `n` will be `double[]`. It is not necessary (or legal) to specify it as `n[]`. It would be legal to explicitly declare it as `double[] n`, but doing so gains nothing in this case.

Lambda Expressions and Variable Capture

Variables defined by the enclosing scope of a lambda expression are accessible within the lambda expression. For example, a lambda expression can use an instance or `static` variable defined by its enclosing class. A lambda expression also has access to `this` (both explicitly and implicitly), which refers to the invoking instance of the lambda expression's enclosing class. Thus, a lambda expression can obtain or set the value of an instance or `static` variable and call a method defined by its enclosing class.

However, when a lambda expression uses a local variable from its enclosing scope, a special situation is created that is referred to as a *variable capture*. In this case, a lambda expression may only use local variables that are *effectively final*. An effectively final variable is one whose value does not change after it is first assigned. There is no need to explicitly declare such a variable as `final`, although doing so would not be an error. (The `this` parameter of an enclosing scope is automatically effectively final, and lambda expressions do not have a `this` of their own.)

It is important to understand that a local variable of the enclosing scope cannot be modified by the lambda expression. Doing so would remove its effectively final status, thus rendering it illegal for capture.

The following program illustrates the difference between effectively final and mutable local variables:

```
// An example of capturing a local variable from the enclosing scope.

interface MyFunc {
    int func(int n);
}

class VarCapture {
    public static void main(String[] args)
    {
        // A local variable that can be captured.
        int num = 10;

        MyFunc myLambda = (n) -> {
            // This use of num is OK. It does not modify num.
            int v = num + n;

            // However, the following is illegal because it attempts
            // to modify the value of num.
            // num++;

            return v;
        };

        // The following line would also cause an error, because
        // it would remove the effectively final status from num.
        // num = 9;
    }
}
```

As the comments indicate, **num** is effectively final and can, therefore, be used inside **myLambda**. However, if **num** were to be modified, either inside the lambda or outside of it, **num** would lose its effectively final status. This would cause an error, and the program would not compile.

It is important to emphasize that a lambda expression can use and modify an instance variable from its invoking class. It just can't use a local variable of its enclosing scope unless that variable is effectively final.

Method References

There is an important feature related to lambda expressions called the *method reference*. A method reference provides a way to refer to a method without executing it. It relates to lambda expressions because it, too, requires a target type context that consists of a compatible functional interface. When evaluated, a method reference also creates an instance of the functional interface.

There are different types of method references. We will begin with method references to **static** methods.

Method References to static Methods

To create a **static** method reference, use this general syntax:

ClassName::methodName

Notice that the class name is separated from the method name by a double colon. The :: is a separator that was added to Java by JDK 8 expressly for this purpose. This method reference can be used anywhere in which it is compatible with its target type.

The following program demonstrates a **static** method reference:

```
// Demonstrate a method reference for a static method.

// A functional interface for string operations.
interface StringFunc {
    String func(String n);
}

// This class defines a static method called strReverse().
class MyStringOps {
    // A static method that reverses a string.
    static String strReverse(String str) {
```

```
        String result = "";
        int i;

        for(i = str.length()-1; i >= 0; i--)
            result += str.charAt(i);

        return result;
    }
}

class MethodRefDemo {

    // This method has a functional interface as the type of
    // its first parameter. Thus, it can be passed any instance
    // of that interface, including a method reference.
    static String stringOp(StringFunc sf, String s) {
        return sf.func(s);
    }

    public static void main(String[] args)
    {
        String inStr = "Lambdas add power to Java";
        String outStr;

        // Here, a method reference to strReverse is passed to stringOp().
        outStr = stringOp(MyStringOps::strReverse, inStr);

        System.out.println("Original string: " + inStr);
        System.out.println("String reversed: " + outStr);
    }
}
```

The output is shown here:

```
Original string: Lambdas add power to Java
String reversed: avaJ ot rewop dda sadbmaL
```

In the program, pay special attention to this line:

```
outStr = stringOp(MyStringOps::strReverse, inStr);
```

Here, a reference to the **static** method **strReverse()**, declared inside **MyStringOps**, is passed as the first argument to **stringOp()**. This works because **strReverse** is compatible with the **StringFunc** functional interface. Thus, the expression **MyStringOps::strReverse** evaluates to a reference to an object in which **strReverse** provides the implementation of **func()** in **StringFunc**.

Method References to Instance Methods

To pass a reference to an instance method on a specific object, use this basic syntax:

objRef::methodName

As you can see, the syntax is similar to that used for a **static** method, except that an object reference is used instead of a class name. Here is the previous program rewritten to use an instance method reference:

```
// Demonstrate a method reference to an instance method

// A functional interface for string operations.
interface StringFunc {
    String func(String n);
}

// Now, this class defines an instance method called strReverse().
class MyStringOps {
    String strReverse(String str) {
        String result = "";
        int i;

        for(i = str.length()-1; i >= 0; i--)
            result += str.charAt(i);

        return result;
    }
}

class MethodRefDemo2 {

    // This method has a functional interface as the type of
    // its first parameter. Thus, it can be passed any instance
    // of that interface, including method references.
    static String stringOp(StringFunc sf, String s) {
        return sf.func(s);
    }

    public static void main(String[] args)
    {
        String inStr = "Lambdas add power to Java";
        String outStr;

        // Create a MyStringOps object.
        MyStringOps strOps = new MyStringOps( );

        // Now, a method reference to the instance method strReverse
        // is passed to stringOp().
        outStr = stringOp(strOps::strReverse, inStr);

        System.out.println("Original string: " + inStr +
                           "\nReversed string: " + outStr);
    }
}
```

This program produces the same output as the previous version.

In the program, notice that **strReverse()** is now an instance method of **MyStringOps**. Inside **main()**, an instance of **MyStringOps** called **strOps** is created. This instance is used to create the method reference to **strReverse** in the call to **stringOp**, as shown again here:

```
outStr = stringOp(strOps::strReverse, inStr);
```

In this example, **strReverse()** is called on the **strOps** object.

It is also possible to handle a situation in which you want to specify an instance method that can be used with any object of a given class—not just a specified object. In this case, you will create a method reference as shown here:

ClassName::instanceMethodName

Here, the name of the class is used instead of a specific object, even though an instance method is specified. With this form, the first parameter of the functional interface matches the invoking object and the second parameter matches the parameter specified by the method. Here is an example. It defines a method called **counter()** that counts the number of objects in an array that satisfy the condition defined by the **func()** method of the **MyFunc** functional interface. In this case, it counts instances of the **HighTemp** class.

```
// Use an instance method reference with different objects.

// A functional interface that takes two reference arguments
// and returns a boolean result.
interface MyFunc<T> {
    boolean func(T v1, T v2);
}

// A class that stores the temperature high for a day.
class HighTemp {
    private int hTemp;

    HighTemp(int ht) { hTemp = ht; }

    // Return true if the invoking HighTemp object has the same
    // temperature as ht2.
    boolean sameTemp(HighTemp ht2) {
        return hTemp == ht2.hTemp;
    }

    // Return true if the invoking HighTemp object has a temperature
    // that is less than ht2.
    boolean lessThanTemp(HighTemp ht2) {
        return hTemp < ht2.hTemp;
    }
}

class InstanceMethWithObjectRefDemo {

    // A method that returns the number of occurrences
    // of an object for which some criteria, as specified by
    // the MyFunc parameter, is true.
    static <T> int counter(T[] vals, MyFunc<T> f, T v) {
```

```
int count = 0;

for(int i=0; i < vals.length; i++)
    if(f.func(vals[i], v)) count++;

return count;
}

public static void main(String[] args)
{
    int count;

    // Create an array of HighTemp objects.
    HighTemp[] weekDayHighs = { new HighTemp(89), new HighTemp(82),
                               new HighTemp(90), new HighTemp(89),
                               new HighTemp(89), new HighTemp(91),
                               new HighTemp(84), new HighTemp(83) };

    // Use counter() with arrays of the class HighTemp.
    // Notice that a reference to the instance method
    // sameTemp() is passed as the second argument.
    count = counter(weekDayHighs, HighTemp::sameTemp,
                    new HighTemp(89));
    System.out.println(count + " days had a high of 89");

    // Now, create and use another array of HighTemp objects.
    HighTemp[] weekDayHighs2 = { new HighTemp(32), new HighTemp(12),
                                new HighTemp(24), new HighTemp(19),
                                new HighTemp(18), new HighTemp(12),
                                new HighTemp(-1), new HighTemp(13) };

    count = counter(weekDayHighs2, HighTemp::sameTemp,
                    new HighTemp(12));
    System.out.println(count + " days had a high of 12");

    // Now, use lessThanTemp() to find days when temperature was less
    // than a specified value.
    count = counter(weekDayHighs, HighTemp::lessThanTemp,
                    new HighTemp(89));
    System.out.println(count + " days had a high less than 89");
```

The output is shown here:

```
Count = counter(weekDayHighs2, HighTemp::lessThanTemp,
                new HighTemp(19));
3 days had a high of 89                                id a high of less than 19";
2 days had a high of 12
3 days had a high less than 89
5 days had a high of less than 19
```

In the program, notice that **HighTemp** has two instance methods: **sameTemp()** and **lessThanTemp()**. The first returns **true** if two **HighTemp** objects contain the same temperature. The second returns **true** if the temperature of the invoking object is less than that of the passed object. Each method has a parameter of type **HighTemp**, and each method returns a **boolean** result. Thus, each is compatible with the **MyFunc** functional interface because the invoking object type can be mapped to the first parameter of **func()** and the argument mapped to **func()**'s second parameter. Thus, when the expression

```
HighTemp::sameTemp
```

is passed to the **counter()** method, an instance of the functional interface **MyFunc** is created in which the parameter type of the first parameter is that of the invoking object of the instance method, which is **HighTemp**. The type of the second parameter is also **HighTemp** because that is the type of the parameter to **sameTemp()**. The same is true for the **lessThanTemp()** method.

One other point: you can refer to the superclass version of a method by use of **super**, as shown here:

```
super::name
```

The name of the method is specified by *name*. Another form is

```
typeName.super::name
```

where *typeName* refers to an enclosing class or super interface.

Method References with Generics

You can use method references with generic classes and/or generic methods. For example, consider the following program:

```
// Demonstrate a method reference to a generic method
// declared inside a non-generic class.

// A functional interface that operates on an array
// and a value, and returns an int result.
interface MyFunc<T> {
    int func(T[] vals, T v);
}

// This class defines a method called countMatching() that
// returns the number of items in an array that are equal
// to a specified value. Notice that countMatching()
// is generic, but MyArrayOps is not.
class MyArrayOps {
    static <T> int countMatching(T[] vals, T v) {
        int count = 0;

        for(int i=0; i < vals.length; i++)
            if(vals[i] == v) count++;

        return count;
    }
}
```

```

class GenericMethodRefDemo {

    // This method has the MyFunc functional interface as the
    // type of its first parameter. The other two parameters
    // receive an array and a value, both of type T.
    static <T> int myOp(MyFunc<T> f, T[] vals, T v) {
        return f.func(vals, v);
    }

    public static void main(String[] args)
    {
        Integer[] vals = { 1, 2, 3, 4, 2, 3, 4, 4, 5 };
        String[] strs = { "One", "Two", "Three", "Two" };
        int count;

        count = myOp(MyArrayOps::<Integer>countMatching, vals, 4);
        System.out.println("vals contains " + count + " 4s");

        count = myOp(MyArrayOps::<String>countMatching, strs, "Two");
        System.out.println("strs contains " + count + " Twos");
    }
}

```

The output is shown here:

```

vals contains 3 4s
strs contains 2 Twos

```

In the program, **MyArrayOps** is a non-generic class that contains a generic method called **countMatching()**. The method returns a count of the elements in an array that match a specified value. Notice how the generic type argument is specified. For example, its first call in **main()**, shown here:

```

count = myOp(MyArrayOps::<Integer>countMatching, vals, 4);

```

passes the type argument **Integer**. Notice that it occurs after the `::`. This syntax can be generalized: When a generic method is specified as a method reference, its type argument comes after the `::` and before the method name. It is important to point out, however, that explicitly specifying the type

argument is not required in this situation (and many others) because the type argument would have been automatically inferred. In cases in which a generic class is specified, the type argument follows the class name and precedes the ::.

Although the preceding examples show the mechanics of using method references, they don't show their real benefits. One place method references can be quite useful is in conjunction with the Collections Framework, which is described later in [Chapter 20](#). However, for completeness, a short, but effective, example that uses a method reference to help determine the largest element in a collection is included here. (If you are unfamiliar with the Collections Framework, return to this example after you have worked through [Chapter 20](#).)

One way to find the largest element in a collection is to use the **max()** method defined by the **Collections** class. For the version of **max()** used here, you must pass a reference to the collection and an instance of an object that implements the **Comparator<T>** interface. This interface specifies how two objects are compared. It defines only one abstract method, called **compare()**, that takes two arguments, each the type of the objects being compared. It must return greater than zero if the first argument is greater than the second, zero if the two arguments are equal, and less than zero if the first object is less than the second.

In the past, to use **max()** with user-defined objects, an instance of **Comparator<T>** had to be obtained by first explicitly implementing it by a class, and then creating an instance of that class. This instance was then passed as the comparator to **max()**. Beginning with JDK 8, it is now possible to simply pass a reference to a comparison method to **max()** because doing so automatically implements the comparator. The following simple example shows the process by creating an **ArrayList** of **MyClass** objects and then finding the one in the list that has the highest value (as defined by the comparison method):

```
// Use a method reference to help find the maximum value in a collection.
import java.util.*;

class MyClass {
    private int val;

    MyClass(int v) { val = v; }

    int getVal() { return val; }
}

class UseMethodRef {
    // A compare() method compatible with the one defined by Comparator<T>.
    static int compareMC(MyClass a, MyClass b) {
        return a.getVal() - b.getVal();
    }

    public static void main(String[] args)
    {
        ArrayList<MyClass> al = new ArrayList<MyClass>();

        al.add(new MyClass(1));
        al.add(new MyClass(4));
        al.add(new MyClass(2));
        al.add(new MyClass(9));
        al.add(new MyClass(3));
        al.add(new MyClass(7));

        // Find the maximum value in al using the compareMC() method.
        MyClass maxValObj = Collections.max(al, UseMethodRef::compareMC);

        System.out.println("Maximum value is: " + maxValObj.getVal());
    }
}
```

The output is shown here:

```
Maximum value is: 9
```

The output is shown here:

```
Maximum value is: 9
```

In the program, notice that **MyClass** neither defines any comparison method of its own, nor does it implement **Comparator**. However, the maximum value of a list of **MyClass** items can still be obtained by calling **max()** because **UseMethodRef** defines the static method **compareMC()**, which is compatible with the **compare()** method defined by **Comparator**. Therefore, there is no need to explicitly implement and create an instance of **Comparator**.

Constructor References

Similar to the way that you can create references to methods, you can create references to constructors. Here is the general form of the syntax that you will use:

classname::new

This reference can be assigned to any functional interface reference that defines a method compatible with the constructor. Here is a simple example:

```
// Demonstrate a Constructor reference.

// MyFunc is a functional interface whose method returns
// a MyClass reference.
interface MyFunc {
    MyClass func(int n);
}

class MyClass {
    private int val;

    // This constructor takes an argument.
    MyClass(int v) { val = v; }

    // This is the default constructor.
    MyClass() { val = 0; }

    // ...

    int getVal() { return val; }
}

class ConstructorRefDemo {
    public static void main(String[] args)
    {
        // Create a reference to the MyClass constructor.
        // Because func() in MyFunc takes an argument, new
        // refers to the parameterized constructor in MyClass,
        // not the default constructor.
        MyFunc myClassCons = MyClass::new;

        // Create an instance of MyClass via that constructor reference.
        MyClass mc = myClassCons.func(100);

        // Use the instance of MyClass just created.
        System.out.println("val in mc is " + mc.getVal());
    }
}
```

The output is shown here:

```
val in mc is 100
```

In the program, notice that the **func()** method of **MyFunc** returns a reference of type **MyClass** and has an **int** parameter. Next, notice that **MyClass** defines two constructors. The first specifies a parameter of type **int**. The second is the default, parameterless constructor. Now, examine the following line:

```
MyFunc myClassCons = MyClass::new;
```

Here, the expression **MyClass::new** creates a constructor reference to a **MyClass** constructor. In this case, because **MyFunc**'s **func()** method takes an **int** parameter, the constructor being referred to is **MyClass(int v)** because it is the one that matches. Also notice that the reference to this constructor is assigned to a **MyFunc** reference called **myClassCons**. After this statement executes, **myClassCons** can be used to create an instance of **MyClass**, as this line shows:

```
MyClass mc = myClassCons.func(100);
```

In essence, **myClassCons** has become another way to call **MyClass(int v)**.

Constructor references to generic classes are created in the same fashion. The only difference is that the type argument can be specified. This works the same as it does for using a generic class to create a method reference: simply specify the type argument after the class name. The following illustrates this by modifying the previous example so that **MyFunc** and **MyClass** are generic:

```
// Demonstrate a constructor reference with a generic class.

// MyFunc is now a generic functional interface.
interface MyFunc<T> {
    MyClass<T> func(T n);
}

class MyClass<T> {
    private T val;

    // A constructor that takes an argument.
    MyClass(T v) { val = v; }

    // This is the default constructor.
    MyClass() { val = null; }

    // ...
    T getVal() { return val; }
}

class ConstructorRefDemo2 {

    public static void main(String[] args)
    {
        // Create a reference to the MyClass<T> constructor.
        MyFunc<Integer> myClassCons = MyClass<Integer>::new;

        // Create an instance of MyClass<T> via that constructor reference.
        MyClass<Integer> mc = myClassCons.func(100);

        // Use the instance of MyClass<T> just created.
        System.out.println("val in mc is " + mc.getVal());
    }
}
```

This program produces the same output as the previous version. The difference is that now both **MyFunc** and **MyClass** are generic. Thus, the sequence that creates a constructor reference can include a type argument (although one is not always needed), as shown here:

```
MyFunc<Integer> myClassCons = MyClass<Integer>::new;
```

Because the type argument **Integer** has already been specified when **myClassCons** is created, it can be used to create a **MyClass<Integer>** object, as the next line shows:

```
MyClass<Integer> mc = myClassCons.func(100);
```

Although the preceding examples demonstrate the mechanics of using a constructor reference, no one would use a constructor reference as just shown because nothing is gained. Furthermore, having what amounts to two names for the same constructor creates a confusing situation (to say the least). However, to give you the flavor of a more practical usage, the following program uses a **static** method, called **myClassFactory()**, that is a factory for objects of any type of **MyFunc** objects. It can be used to create any type of object that has a constructor compatible with its first parameter.

```
// Implement a simple class factory using a constructor reference.

interface MyFunc<R, T> {
    R func(T n);
}

// A simple generic class.
class MyClass<T> {
    private T val;

    // A constructor that takes an argument.
    MyClass(T v) { val = v; }
```

```
// The default constructor. This constructor
// is NOT used by this program.
MyClass() { val = null; }
// ...

T getVal() { return val; }

}

// A simple, non-generic class.
class MyClass2 {
    String str;

    // A constructor that takes an argument.
    MyClass2(String s) { str = s; }

    // The default constructor. This
    // constructor is NOT used by this program.
    MyClass2() { str = ""; }

    // ...

    String getVal() { return str; }
}

class ConstructorRefDemo3 {

    // A factory method for class objects. The class must
    // have a constructor that takes one parameter of type T.
    // R specifies the type of object being created.
    static <R,T> R myClassFactory(MyFunc<R, T> cons, T v) {
        return cons.func(v);
    }

    public static void main(String[] args)
    {
        // Create a reference to a MyClass constructor.
        // In this case, new refers to the constructor that
        // takes an argument.
        MyFunc<MyClass<Double>, Double> myClassCons = MyClass<Double>::new;

        // Create an instance of MyClass by use of the factory method.
        MyClass<Double> mc = myClassFactory(myClassCons, 100.1);
    }
}
```

```

        // Use the instance of MyClass just created.
        System.out.println("str in mc2 is " + mc2.getVal( ));
    }
}

// Now, create a different class by use of myClassFactory().
The output is shown here:
MyFunc<MyClass2, String> myClassCons2 = MyClass2::new;

val in mc is 100.1
str in mc2 is Lambda : of MyClass2 by use of the factory method.
MyClass2 mc2 = myClassFactory(myClassCons2, "Lambda");

```

As you can see, **myClassFactory()** is used to create objects of type **MyClass<Double>** and **MyClass2**. Although both classes differ—for example, **MyClass** is generic and **MyClass2** is not—both can be created by **myClassFactory()** because they both have constructors that are compatible with **func()** in **MyFunc**. This works because **myClassFactory()** is passed the constructor for the object that it builds. You might want to experiment with this program a bit, trying different classes that you create. Also try creating instances of different types of **MyClass** objects. As you will see, **myClassFactory()** can create any type of object whose class has a constructor that is compatible with **func()** in **MyFunc**. Although this example is quite simple, it hints at the power that constructor references bring to Java.

Before moving on, it is important to mention a second form of the constructor reference syntax that is used for arrays. To create a constructor reference for an array, use this construct:

type[::new

Here, *type* specifies the type of object being created. For example, assuming the form of **MyClass** as shown in the first constructor reference example (**ConstructorRefDemo**) and given the **MyArrayCreator** interface shown here:

```

interface MyArrayCreator<T> {
    T func(int n);
}

```

the following creates a two-element array of **MyClass** objects and gives each element an initial value:

```

MyArrayCreator<MyClass [] > mcArrayCons = MyClass [] ::new;
MyClass [] a = mcArrayCons.func(2);
a[0] = new MyClass(1);
a[1] = new MyClass(2);

```

Here, the call to **func(2)** causes a two-element array to be created. In general, a functional interface must contain a method that takes a single **int** parameter if it is to be used to refer to an array constructor.

Predefined Functional Interfaces

Up to this point, the examples in this chapter have defined their own functional interfaces so that the fundamental concepts behind lambda expressions and functional interfaces could be clearly illustrated. However, in many cases, you won't need to define your own functional interface because the package called **java.util.function** provides several predefined ones. Although we will look at them more closely in Part II, here is a sampling:

Interface	Purpose
UnaryOperator<T>	Apply a unary operation to an object of type T and return the result, which is also of type T. Its method is called apply() .
BinaryOperator<T>	Apply an operation to two objects of type T and return the result, which is also of type T. Its method is called apply() .
Consumer<T>	Apply an operation on an object of type T. Its method is called accept() .
Supplier<T>	Return an object of type T. Its method is called get() .
Function<T, R>	Apply an operation to an object of type T and return the result as an object of type R. Its method is called apply() .
Predicate<T>	Determine if an object of type T fulfills some constraint. Return a boolean value that indicates the outcome. Its method is called test() .

The following program shows the **Function** interface in action by using it to rework the earlier example called **BlockLambdaDemo** that demonstrated block lambdas by implementing a factorial example. That example created its own functional interface called **NumericFunc**, but the

built-in **Function** interface could have been used, as this version of the program illustrates:

```
// Use the Function built-in functional interface.

// Import the Function interface.
import java.util.function.Function;

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

        // This block lambda computes the factorial of an int value.
        // This time, Function is the functional interface.
        Function<Integer, Integer> factorial = (n) ->  {
            int result = 1;
            for(int i=1; i <= n; i++)
                result = i * result;
            return result;
        };

        System.out.println("The factorial of 3 is " + factorial.apply(3));
        System.out.println("The factorial of 5 is " + factorial.apply(5));
    }
}
```

It produces the same output as previous versions of the program.

CHAPTER

16

Modules

JDK 9 introduced a new and important feature called *modules*. Modules give you a way to describe the relationships and dependencies of the code that comprises an application. Modules also let you control which parts of a module are accessible to other modules and which are not. Through the use of modules you can create more reliable, scalable programs.

As a general rule, modules are most helpful to large applications because they help reduce the management complexity often associated with a large software system. However, small programs also benefit from modules because the Java API library has now been organized into modules. Thus, it is now possible to specify which parts of the API are required by your program and which are not. This makes it possible to deploy programs with a smaller run-time footprint, which is especially important when creating code for small devices, such as those intended to be part of the Internet of Things (IoT).

Support for modules is provided both by language elements, including several keywords, and by enhancements to **javac**, **java**, and other JDK tools. Furthermore, new tools and file formats were introduced. As a result, the JDK and the run-time system were substantially upgraded to support modules. In short, modules constitute a major addition to, and evolution of, the Java language.

Module Basics

In its most fundamental sense, a *module* is a grouping of packages and resources that can be collectively referred to by the module's name. A

module declaration specifies the name of a module and defines the relationship a module and its packages have to other modules. Module declarations are program statements in a Java source file and are supported by several module-related keywords. They are shown here:

exports	module	open	opens
provides	requires	to	transitive
uses	with		

It is important to understand that these keywords are recognized *as keywords* only in the context of a module declaration. Otherwise, they are interpreted as identifiers in other situations. Thus, the keyword **module** could, for example, also be used as a parameter name, although such a use is certainly not recommended. However, making the module-related keywords context-sensitive prevents problems with pre-existing code that may use one or more of them as identifiers.

A module declaration is contained in a file called **module-info.java**. Thus, a module is defined in a Java source file. This file is then compiled by **javac** into a class file and is known as its *module descriptor*. The **module-info.java** file must contain only a module definition. It cannot contain other types of declarations.

A module declaration begins with the keyword **module**. Here is its general form:

```
module moduleName {  
    // module definition  
}
```

The name of the module is specified by *moduleName*, which must be a valid Java identifier or a sequence of identifiers separated by periods. The module definition is specified within the braces. Although a module definition may be empty (which results in a declaration that simply names the module), typically it specifies one or more clauses that define the characteristics of the module.

A Simple Module Example

At the foundation of a module's capabilities are two key features. The first is a module's ability to specify that it requires another module. In other words, one module can specify that it *depends* on another. A dependence relationship is specified by use of a **requires** statement. By default, the presence of the required module is checked at both compile time and at run time. The second key feature is a module's ability to control which, if any, of its packages are accessible by another module. This is accomplished by use of the **exports** keyword. The public and protected types within a package are accessible to other modules only if they are explicitly exported. Here we will develop an example that introduces both of these features.

The following example creates a modular application that demonstrates some simple mathematical functions. Although this application is purposely very small, it illustrates the core concepts and procedures required to create, compile, and run module-based code. Furthermore, the general approach shown here also applies to larger, real-world applications. It is strongly recommended that you work through the example on your computer, carefully following each step.

NOTE This chapter shows the process of creating, compiling, and running module-based code by use of the command-line tools. This approach has two advantages. First, it works for all Java programmers because no IDE is required. Second, it very clearly shows the fundamentals of the module system, including how it utilizes directories. To follow along, you will need to manually create a number of directories and ensure that each file is placed in its proper directory. As you might expect, when creating real-world, module-based applications, you will likely find a module-aware IDE easier to use because, typically, it will automate much of the process. However, learning the fundamentals of modules using the command-line tools ensures that you have a solid understanding of the topic.

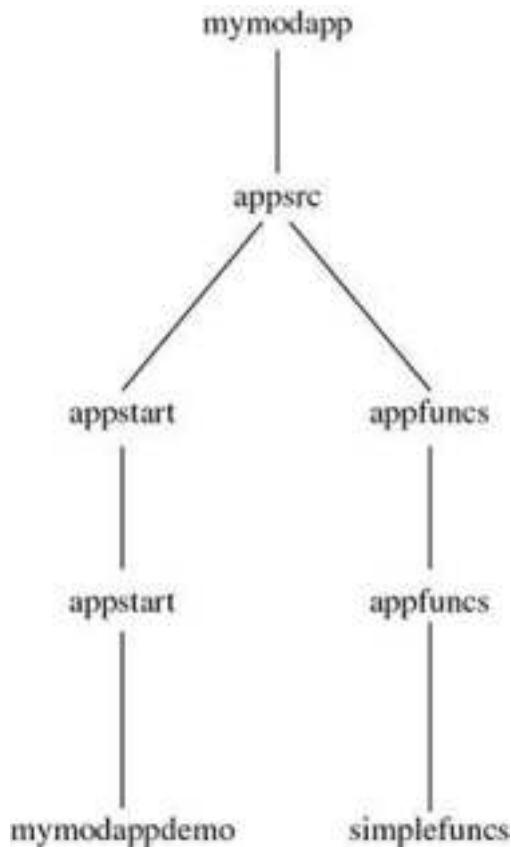
The application defines two modules. The first module is called **appstart**. It contains a package called **appstart.mymodappdemo** that defines the application's entry point in a class called **MyModAppDemo**. Thus, **MyModAppDemo** contains the application's **main()** method. The second module is called **appfuncs**. It contains a package called **appfuncs.simplefuncs** that includes the class **SimpleMathFuncs**. This class defines three static methods that implement some simple mathematical functions. The entire application will be contained in a directory tree that begins at **mymodapp**.

Before continuing, a few words about module names are appropriate. First, in the examples that follow, the name of a module (such as **appfuncs**) is the prefix of the name of a package (such as **appfuncs.simplefuncs**) that it contains. This is *not* required, but it's used here as a way of clearly indicating to what module a package belongs. In general, when learning about and experimenting with modules, short, simple names, such as those used in this chapter, are helpful, and you can use any sort of convenient names that you like. However, when creating modules suitable for distribution, you must be careful with the names you choose because you will want those names to be unique. At the time of this writing, the suggested way to achieve this is to use the reverse domain name method. In this method, the reverse domain name of the domain that "owns" the project is used as a prefix for the module. For example, a project associated with **hotburgers.com** would use **com.hotburgers** as the module prefix. (The same goes for package names.) Because naming conventions may evolve over time, you will want to check the Java documentation for current recommendations.

Let's now begin. Start by creating the necessary source code directories by following these steps:

1. Create a directory called **mymodapp**. This is the top-level directory for the entire application.
2. Under **mymodapp**, create a subdirectory called **apps**. This is the top-level directory for the application's source code.
3. Under **apps**, create the subdirectory **appstart**. Under this directory, create a subdirectory also called **appstart**. Under this directory, create the directory **mymodappdemo**. Thus, beginning with **apps**, you will have created this tree:
`apps\appstart\appstart\mymodappdemo`
4. Also under **apps**, create the subdirectory **appfuncs**. Under this directory, create a subdirectory also called **appfuncs**. Under this directory, create the directory called **simplefuncs**. Thus, beginning with **apps**, you will have created this tree:
`apps\appfuncs\appfuncs\simplefuncs`

Your directory tree should look like that shown here.



After you have set up these directories, you can create the application's source files.

This example will use four source files. Two are the source files that define the application. The first is **SimpleMathFuncs.java**, shown here. Notice that **SimpleMathFuncs** is packaged in **appfuncs.simplefuncs**.

```
// Some simple math functions.

package appfuncs.simplefuncs;

public class SimpleMathFuncs {

    // Determine if a is a factor of b.
    public static boolean isFactor(int a, int b) {
        if((b%a) == 0) return true;
        return false;
    }

    // Return the smallest positive factor that a and b have in common.
    public static int lcf(int a, int b) {
        // Factor using positive values.
        a = Math.abs(a);
        b = Math.abs(b);

        int min = a < b ? a : b;

        for(int i = 2; i <= min/2; i++) {
            if(isFactor(i, a) && isFactor(i, b))
                return i;
        }
    }
}
```

```

        return 1;
    }

    // Return the largest positive factor that a and b have in common.
    public static int gcf(int a, int b) {
        // Factor using positive values.
        a = Math.abs(a);
        b = Math.abs(b);

        int min = a < b ? a : b;

        for(int i = min/2; i >= 2; i--) {
            if(isFactor(i, a) && isFactor(i, b))
                return i;
        }

        return 1;
    }
}

```

SimpleMathFuncs defines three simple **static** math functions. The first, **isFactor()**, returns true if **a** is a factor of **b**. The **lcf()** method returns the smallest factor common to both **a** and **b**. In other words, it returns the least common factor of **a** and **b**. The **gcf()** method returns the greatest common factor of **a** and **b**. In both cases, 1 is returned if no common factors are found. This file must be put in the following directory:

apps\src\appfuncs\appfuncs\simplefuncs

This is the **appfuncs.simplefuncs** package directory.

The second source file is **MyModAppDemo.java**, shown next. It uses the methods in **SimpleMathFuncs**. Notice that it is packaged in **appstart.mymodappdemo**. Also note that it imports the **SimpleMathFuncs** class because it depends on **SimpleMathFuncs** for its operation.

```
// Demonstrate a simple module-based application.
package appstart.mymodappdemo;

import appfuncs.simplefuncs.SimpleMathFuncs;

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

        if(SimpleMathFuncs.isFactor(2, 10))
            System.out.println("2 is a factor of 10");

        System.out.println("Smallest factor common to both 35 and 105 is " +
                           SimpleMathFuncs.lcf(35, 105));

        System.out.println("Largest factor common to both 35 and 105 is " +
                           SimpleMathFuncs.gcf(35, 105));
    }
}
```

This file must be put in the following directory:

apps\appstart\appstart\mymodappdemo

This is the directory for the **appstart.mymodappdemo** package.

Next, you will need to add **module-info.java** files for each module. These files contain the module definitions. First, add this one, which defines the **appfuncs** module:

```
// Module definition for the functions module.
module appfuncs {
    // Exports the package appfuncs.simplefuncs.
    exports appfuncs.simplefuncs;
}
```

Notice that **appfuncs** exports the package **appfuncs.simplefuncs**, which makes it accessible to other modules. This file must be put into this directory:

apps\appfuncs

Thus, it goes in the **appfuncs** module directory, which is above the package directories.

Finally, the **module-info.java** file for the **appstart** module is shown next. Notice that **appstart** requires the module **appfuncs**.

```
// Module definition for the main application module.  
module appstart {  
    // Requires the module appfuncs.  
    requires appfuncs;  
}
```

This file must be put into its module directory:

```
apps\src\appstart
```

Before examining the **requires**, **exports**, and **module** statements more closely, let's first compile and run this example. Be sure that you have correctly created the directories and entered each file into its proper directory, as just explained.

Compile and Run the First Module Example

Beginning with JDK 9, **javac** has been updated to support modules. Thus, like all other Java programs, module-based programs are compiled using **javac**. The process is easy, with the primary difference being that you will usually explicitly specify a *module path*. A module path tells the compiler where the compiled files will be located. When following along with this example, be sure that you execute the **javac** commands from the **mymodapp** directory in order for the paths to be correct. Recall that **mymodapp** is the top-level directory for the entire module application.

To begin, compile **SimpleMathFuncs.java** using this command:

```
javac -d appmodules\appfuncs  
      apps\src\appfuncs\appfuncs\simplefuncs\SimpleMathFuncs.java
```

Remember, this command *must be* executed from the **mymodapp** directory. Notice the use of the **-d** option. This tells **javac** where to put the output **.class** file. For the examples in this chapter, the top of the directory tree for

compiled code is **appmodules**. This command will create the output package directories for **appfuncs.simplefuncs** under **appmodules\appfuncs** as needed.

Next, here is the **javac** command that compiles the **module-info.java** file for the **appfuncs** module:

```
javac -d appmodules\appfuncs appsrc\appfuncs\module-info.java
```

This puts the **module-info.class** file into the **appmodules\appfuncs** directory.

Although the preceding two-step process works, it was shown primarily for the sake of discussion. It is usually easier to compile a module's **module-info.java** file and its source files in one command line. Here, the preceding two **javac** commands are combined into one:

```
javac -d appmodules\appfuncs appsrc\appfuncs\module-info.java  
      appsrc\appfuncs\appfuncs\simplefuncs\SimpleMathFuncs.java
```

In this case, each compiled file is put in its proper module or package directory.

Now, compile **module-info.java** and **MyModAppDemo.java** for the **appstart** module, using this command:

```
javac --module-path appmodules -d appmodules\appstart  
      appsrc\appstart\module-info.java  
      appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java
```

Notice the **--module-path** option. It specifies the module path, which is the path on which the compiler will look for the user-defined modules required by the **module-info.java** file. In this case, it will look for the **appfuncs** module because it is needed by the **appstart** module. Also, notice that it specifies the output directory as **appmodules\appstart**. This means that the **module-info.class** file will be in the **appmodules\appstart** module directory and **MyModAppDemo.class** will be in the **appmodules\appstart\appstart\mymodappdemo** package directory.

Once you have completed the compilation, you can run the application with this **java** command:

```
java --module-path appmodules -m  
appstart/appstart.mymodappdemo.MyModAppDemo
```

Here, the **--module-path** option specifies the path to the application's modules. As mentioned, **appmodules** is the directory at the top of the compiled modules tree. The **-m** option specifies the class that contains the entry point of the application and, in this case, the name of the class that contains the **main()** method. When you run the program, you will see the following output:

```
2 is a factor of 10  
Smallest factor common to both 35 and 105 is 5  
Largest factor common to both 35 and 105 is 7
```

A Closer Look at requires and exports

The preceding module-based example relies on the two foundational features of the module system: the ability to specify a dependence and the ability to satisfy that dependence. These capabilities are specified through the use of the **requires** and **exports** statements within a **module** declaration. Each merits a closer examination at this time.

Here is the form of the **requires** statement used in the example:

```
requires moduleName;
```

Here, *moduleName* specifies the name of a module that is required by the module in which the **requires** statement occurs. This means that the required module must be present in order for the current module to compile. In the language of modules, the current module is said to *read* the module specified in the **requires** statement. When more than one module is required, it must be specified in its own **requires** statement. Thus, a module declaration may include several different **requires** statements. In general, the **requires** statement gives you a way to ensure that your program has access to the modules that it needs.

Here is the general form of the **exports** statement used in the example:

```
exports packageName;
```

Here, *packageName* specifies the name of the package that is exported by the module in which this statement occurs. A module can export as many packages as needed, with each one specified in a separate **exports** statement. Thus, a module may have several **exports** statements.

When a module exports a package, it makes all of the public and protected types in the package accessible to other modules. Furthermore, the public and protected members of those types are also accessible. However, if a package within a module is not exported, then it is private to that module, including all of its public types. For example, even though a class is declared as **public** within a package, if that package is not explicitly exported by an **exports** statement, then that class is not accessible to other modules. It is important to understand that the public and protected types of a package, whether exported or not, are always accessible within that package's module. The **exports** statement simply makes them accessible to outside modules. Thus, any nonexported package is only for the internal use of its module.

The key to understanding **requires** and **exports** is that they work together. If one module depends on another, then it must specify that dependence with **requires**. The module on which another depends must explicitly export (i.e., make accessible) the packages that the dependent module needs. If either side of this dependence relationship is missing, the dependent module will not compile. As it relates to the foregoing example, **MyModAppDemo** uses the functions in **SimpleMathFuncs**. As a result, the **appstart** module declaration contains a **requires** statement that names the **appfuncs** module. The **appfuncs** module declaration exports the **appfuncs.simplefuncs** package, thus making the public types in the **SimpleMathFuncs** class available. Since both sides of the dependence relationship have been fulfilled, the application can compile and run. If either is missing, the compilation will fail.

It is important to emphasize that **requires** and **exports** statements must occur only within a **module** statement. Furthermore, a **module** statement must occur by itself in a file called **module-info.java**.

java.base and the Platform Modules

As mentioned at the start of this chapter, beginning with JDK 9, the Java API packages have been incorporated into modules. In fact, the

modularization of the API is one of the primary benefits realized by the addition of the modules. Because of their special role, the API modules are referred to as *platform modules*, and their names all begin with the prefix **java**. Here are some examples: **java.base**, **java.desktop**, and **java.xml**. By modularizing the API, it becomes possible to deploy an application with only the packages that it requires, rather than the entire Java Runtime Environment (JRE). Because of the size of the full JRE, this is a very important improvement.

The fact that all of the Java API library packages are now in modules gives rise to the following question: How can the **main()** method in **MyModAppDemo** in the preceding example use **System.out.println()** without specifying a **requires** statement for the module that contains the **System** class? Obviously, the program will not compile and run unless **System** is present. The same question also applies to the use of the **Math** class in **SimpleMathFuncs**. The answer to this question is found in **java.base**.

Of the platform modules, the most important is **java.base**. It includes and exports those packages fundamental to Java, such as **java.lang**, **java.io**, and **java.util**, among many others. Because of its importance, **java.base** is *automatically accessible* to all modules. Furthermore, all other modules automatically require **java.base**. There is no need to include a **requires java.base** statement in a module declaration. (As a point of interest, it is not wrong to explicitly specify **java.base**, it's just not necessary.) Thus, in much the same way that **java.lang** is automatically available to all programs without the use of an **import** statement, the **java.base** module is automatically accessible to all module-based programs without explicitly requesting it.

Because **java.base** contains the **java.lang** package, and **java.lang** contains the **System** class, **MyModAppDemo** in the preceding example can automatically use **System.out.println()** without an explicit **requires** statement. The same applies to the use of the **Math** class in **SimpleMathFuncs**, because the **Math** class is also in **java.lang**. As you will see when you begin to create your own module-based applications, many of the API classes you will commonly need are in the packages included in **java.base**. Thus, the automatic inclusion of **java.base** simplifies the creation of module-based code because Java's core packages are automatically accessible.

One last point: Beginning with JDK 9, the documentation for the Java API now tells you the name of the module in which a package is contained. If the module is **java.base**, then you can use the contents of that package directly. Otherwise, your module declaration must include a **requires** clause for the desired module.

Legacy Code and the Unnamed Module

Another question may have occurred to you when working through the first sample module program. Because Java now supports modules, and the API packages are also contained in modules, why do all of the other programs in the preceding chapters compile and run without error even though they do not use modules? More generally, since there is now over 20 years of Java code in existence and (at the time of this writing) the vast majority of that code does not use modules, how is it possible to compile, run, and maintain that legacy code with a JDK 9 or later compiler? Given Java's original philosophy of "write once, run everywhere," this is a very important question because backward capability must be maintained. As you will see, Java answers this question by providing an elegant, nearly transparent means of ensuring backward compatibility with pre-existing code.

Support for legacy code is provided by two key features. The first is the *unnamed module*. When you use code that is not part of a named module, it automatically becomes part of the unnamed module. The unnamed module has two important attributes. First, all of the packages in the unnamed module are automatically exported. Second, the unnamed module can access any and all other modules. Thus, when a program does not use modules, all API modules in the Java platform are automatically accessible through the unnamed module.

The second key feature that supports legacy code is the automatic use of the class path, rather than the module path. When you compile a program that does not use modules, the class path mechanism is employed, just as it has been since Java's original release. As a result, the program is compiled and run in the same way it was prior to the advent of modules.

Because of the unnamed module and the automatic use of the class path, there was no need to declare any modules for the sample programs shown elsewhere in this book. They run properly whether you compile them with a

modern compiler or an earlier one, such as JDK 8. Thus, even though modules are a feature that has significant impact on Java, compatibility with legacy code is maintained. This approach also provides a smooth, nonintrusive, nondisruptive transition path to modules. Thus, it enables you to move a legacy application to modules at your own pace. Furthermore, it allows you to avoid the use of modules when they are not needed.

Before moving on, an important point needs to be made. For the types of sample programs used elsewhere in this book, and for sample programs in general, there is no benefit in using modules. Modularizing them would simply add clutter and complicate them for no reason or benefit. Furthermore, for many simple programs, there is no need to contain them in modules. For the reasons stated at the start of this chapter, modules are often of the greatest benefit when creating commercial programs. Therefore, no examples outside this chapter will use modules. This also allows the examples to be compiled and run in a pre-JDK 9 environment, which is important to readers using an older version of Java. Thus, except for the examples in this chapter, the examples in this book work for both pre-module and post-module JDKs.

Exporting to a Specific Module

The basic form of the **exports** statement makes a package accessible to any and all other modules. This is often exactly what you want. However, in some specialized development situations, it can be desirable to make a package accessible to only a *specific set* of modules, not *all* other modules. For example, a library developer might want to export a support package to certain other modules within the library, but not make it available for general use. Adding a **to** clause to the **exports** statement provides a means by which this can be accomplished.

In an **exports** statement, the **to** clause specifies a list of one or more modules that have access to the exported package. Furthermore, only those modules named in the **to** clause will have access. In the language of modules, the **to** clause creates what is known as a *qualified export*.

The form of **exports** that includes **to** is shown here:

```
exports packageName to moduleNames;
```

Here, *moduleNames* is a comma-separated list of modules to which the exporting module grants access.

You can try the **to** clause by changing the **module-info.java** file for the **appfuncs** module, as shown here:

```
// Module definition that uses a to clause.  
module appfuncs {  
    // Exports the package appfuncs.simplefuncs to appstart.  
    exports appfuncs.simplefuncs to appstart;  
}
```

Now, **simplefuncs** is exported only to **appstart** and to no other modules. After making this change, you can recompile the application by using this **javac** command:

```
javac -d appmodules --module-source-path appsrc  
    appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java
```

After compiling, you can run the application as shown earlier.

This example also uses another module-related feature. Look closely at the preceding **javac** command. First, notice that it specifies the **--module-source-path** option. The module source path specifies the top of the module source tree. The **--module-source-path** option automatically compiles the files in the tree under the specified directory, which is **appsrc** in this example. The **--module-source-path** option must be used with the **-d** option to ensure that the compiled modules are stored in their proper directories under **appmodules**. This form of **javac** is called *multi-module mode* because it enables more than one module to be compiled at a time. The multi-module compilation mode is especially helpful here because the **to** clause refers to a specific module, and the requiring module must have access to the exported package. Thus, in this case, both **appstart** and **appfuncs** are needed to avoid warnings and/or errors during compilation. Multi-module mode avoids this problem because both modules are being compiled at the same time.

The multi-module mode of **javac** has another advantage. It automatically finds and compiles all source files for the application, creating the necessary output directories. Because of the advantages that

multi-module compilation mode offers, it will be used for the subsequent examples.

NOTE As a general rule, qualified export is a special case feature. Most often, your modules will either provide unqualified export of a package or not export the package at all, keeping it inaccessible. As such, qualified export is discussed here primarily for the sake of completeness. Also, qualified export by itself does not prevent the exported package from being misused by malicious code in a module that masquerades as the targeted module. The security techniques required to prevent this from happening are beyond the scope of this book. Consult the Oracle documentation for details on security in this regard, and Java security details in general.

Using requires transitive

Consider a situation in which there are three modules, A, B, and C, that have the following dependences:

- A requires B.
- B requires C.

Given this situation, it is clear that since A depends on B and B depends on C, A has an indirect dependence on C. As long as A does not directly use any of the contents of C, then you can simply have A require B in its module-info file, and have B export the packages required by A in its module-info file, as shown here:

```
// A's module-info file:  
module A {  
    requires B;  
}  
  
// B's module-info file.  
module B {  
    exports somepack;  
    requires C;  
}
```

Here, *somepack* is a placeholder for the package exported by B and used by A. Although this works as long as A does not need to use anything defined in C, a problem occurs if A *does* want to access a type in C. In this case, there are two solutions.

The first solution is to simply add a **requires C** statement to A's file, as shown here:

```
// A's module-info file updated to explicitly require C:  
module A {  
    requires B;  
    requires C; // also require C  
}
```

This solution certainly works, but if B will be used by many modules, you must add **requires C** to all module definitions that require B. This is not only tedious, it is also error prone. Fortunately, there is a better solution. You can create an *implied dependence* on C. Implied dependence is also referred to as *implied readability*.

To create an implied dependence, add the **transitive** keyword after **requires** in the clause that requires the module upon which an implied readability is needed. In the case of this example, you would change B's module-info file as shown here:

```
// B's module-info file.  
module B {  
    exports somepack;  
    requires transitive C;  
}
```

Here, C is now required as transitive. After making this change, any module that depends on B will also, automatically, depend on C. Thus, A would automatically have access to C.

You can experiment with **requires transitive** by reworking the preceding modular application example so that the **isFactor()** method is removed from the **SimpleMathFuncs** class in the **appfuncs.simplefuncs** package and put into a new class, module, and package. The new class will be called **SupportFuncs**, the module will be called **appsupport**, and the package will be called **appsupport.supportfuncs**. The **appfuncs** module will then add a dependence on the **appsupport** module by use of **requires transitive**. This will enable both the **appfuncs** and **appstart** modules to access it without **appstart** having to provide its own **requires** statement. This works because **appstart** receives access to it through an **appfuncs**

requires transitive statement. The following describes the process in detail.

To begin, create the source directories that support the new **appsupport** module. First, create **appsupport** under the **appsrc** directory. This is the module directory for the support functions. Under **appsupport**, create the package directory by adding the **appsupport** subdirectory followed by the **supportfuncs** subdirectory. Thus, the directory tree for **appsupport** should now look like this:

```
appsrc\appsupport\appsupport\supportfuncs
```

Once the directories have been established, create the **SupportFuncs** class. Notice that **SupportFuncs** is part of the **appsupport.supportfuncs** package. Therefore, you must put it in the **appsupport.supportfuncs** package directory.

```
// Support functions.

package appsupport.supportfuncs;

public class SupportFuncs {

    // Determine if a is a factor of b.
    public static boolean isFactor(int a, int b) {
        if((b%a) == 0) return true;
        return false;
    }
}
```

Notice that **isFactor()** is now part of **SupportFuncs**, rather than **SimpleMathFuncs**.

Next, create the **module-info.java** file for the **appsupport** module and put it in the **appsrc\appsupport** directory.

```
// Module definition for appsupport.
module appsupport {
    exports appsupport.supportfuncs;
}
```

As you can see, it exports the **appsupport.supportfuncs** package.

Because **isFactor()** is now part of **Supportfuncs**, remove it from **SimpleMathFuncs**. Thus, **SimpleMathFuncs.java** will now look like this:

```
// Some simple math functions, with isFactor() removed.

package appfuncs.simplefuncs;
import appsupport.supportfuncs.SupportFuncs;
```

```
public class SimpleMathFuncs {

    // Return the smallest positive factor that a and b have in common.
    public static int lcf(int a, int b) {
        // Factor using positive values.
        a = Math.abs(a);
        b = Math.abs(b);

        int min = a < b ? a : b;

        for(int i = 2; i <= min/2; i++) {
            if(SupportFuncs.isFactor(i, a) && SupportFuncs.isFactor(i, b))
                return i;
        }

        return 1;
    }

    // Return the largest positive factor that a and b have in common.
    public static int gcf(int a, int b) {
        // Factor using positive values.
        a = Math.abs(a);
        b = Math.abs(b);

        int min = a < b ? a : b;

        for(int i = min/2; i >= 2; i--) {
            if(SupportFuncs.isFactor(i, a) && SupportFuncs.isFactor(i, b))
                return i;
        }

        return 1;
    }
}
```

Notice that now the **SupportFuncs** class is imported and calls to **isFactor()** are referred to through the class name **SupportFuncs**.

Next, change the **module-info.java** file for **appfuncs** so that in its **requires** statement, **appsupport** is specified as **transitive**, as shown here:

```
// Module definition for appfuncs.  
module appfuncs {  
    // Exports the package appfuncs.simplefuncs.  
    exports appfuncs.simplefuncs;  
  
    // Requires appsupport and makes it transitive.  
    requires transitive appsupport;  
}
```

Because **appfuncs** requires **appsupport** as **transitive**, there is no need for the **module-info.java** file for **appstart** to also require it. Its dependence on **appsupport** is implied. Thus, no changes to the **module-info.java** file for **appstart** are needed.

Finally, update **MyModAppDemo.java** to reflect these changes. Specifically, it must now import the **SupportFuncs** class and specify it when invoking **isFactor()**, as shown here:

```
// Updated to use SupportFuncs.  
package appstart.mymodappdemo;  
  
import appfuncs.simplefuncs.SimpleMathFuncs;  
import appsupport.supportfuncs.SupportFuncs;  
  
public class MyModAppDemo {  
    public static void main(String[] args) {  
  
        // Now, isFactor() is referred to via SupportFuncs,  
        // not SimpleMathFuncs.  
        if(SupportFuncs.isFactor(2, 10))  
            System.out.println("2 is a factor of 10");  
  
        System.out.println("Smallest factor common to both 35 and 105 is " +  
                           SimpleMathFuncs.lcf(35, 105));  
  
        System.out.println("Largest factor common to both 35 and 105 is " +  
                           SimpleMathFuncs.gcf(35, 105));  
    }  
}
```

Once you have completed all of the preceding steps, you can recompile the entire program using this multi-module compilation command:

```
javac -d appmodules --module-source-path appsrc  
      appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java
```

As explained earlier, the multi-module compilation will automatically create the parallel module subdirectories, under the **appmodules** directory.

You can run the program using this command:

```
java --module-path appmodules -m  
      appstart/appstart.mymodappdemo.MyModAppDemo
```

It will produce the same output as the previous version. However, this time three different modules are required.

To prove that the **transitive** modifier is actually required by the application, remove it from the **module-info.java** file for **appfuncs**. Then, try to compile the program. As you will see, an error will result because **appsupport** is no longer accessible by **appstart**.

Here is another experiment. In the module-info file for **appsupport**, try exporting the **appsupport.supportfuncs** package to only **appfuncs** by use of a qualified export, as shown here:

```
exports appsupport.supportfuncs to appfuncs;
```

Next, attempt to compile the program. As you see, the program will not compile because now the support function **isFactor()** is not available to **MyModAppDemo**, which is in the **appstart** module. As explained previously, a qualified export restricts access to a package to only those modules specified by the **to** clause.

One final point: because of a special exception in the Java language syntax, in a **requires** statement, if **transitive** is immediately followed by a separator (such as a semicolon), it is interpreted as an identifier (for example, as a module name) rather than a keyword.

Use Services

In programming, it is often useful to separate *what* must be done from *how* it is done. As you learned in [Chapter 9](#), one way this is accomplished in Java is through the use of interfaces. The interface specifies the *what*, and the implementing class specifies the *how*. This concept can be expanded so that the implementing class is provided by code that is outside your program, through the use of a *plug-in*. Using such an approach, the capabilities of an application can be enhanced, upgraded, or altered by simply changing the plug-in. The core of the application itself remains unchanged. One way that Java supports a pluggable application architecture is through the use of *services* and *service providers*. Because of their importance, especially in large, commercial applications, Java's module system provides support for them.

Before we begin, it is necessary to state that applications that use services and service providers are typically fairly sophisticated. Therefore, you may find that you do not often need the service-based module features. However, because support for services constitutes a rather significant part of the module system, it is important that you have a general understanding of how these features work. Also, a simple example is presented that illustrates the core techniques needed to use them.

Service and Service Provider Basics

In Java, a *service* is a program unit whose functionality is defined by an interface or abstract class. Thus, a service specifies in a general way some form of program activity. A concrete implementation of a service is supplied by a *service provider*. In other words, a service defines the form of some action, and the service provider supplies that action.

As mentioned, services are often used to support a pluggable architecture. For example, a service might be used to support the translation of one language into another. In this case, the service supports translation in general. The service provider supplies a specific translation, such as German to English or French to Chinese. Because all service providers implement the same interface, different translators can be used to translate different languages without having to change the core of the application. You can simply change the service provider.

Service providers are supported by the **ServiceLoader** class. **ServiceLoader** is a generic class packaged in **java.util**. It is declared like

this:

```
class ServiceLoader<S>
```

Here, **S** specifies the service type. Service providers are loaded by the **load()** method. It has several forms; the one we will use is shown here:

```
public static <S> ServiceLoader<S> load(Class <S> serviceType)
```

Here, *serviceType* specifies the **Class** object for the desired service type. Recall that **Class** is a class that encapsulates information about a class. There are a variety of ways to obtain a **Class** instance. The way we will use here involves a class literal. Recall that a class literal has this general form:

className.class

Here, *className* specifies the name of the class.

When **load()** is called, it returns a **ServiceLoader** instance for the application. This object supports iteration and can be cycled through by use of a for-each **for** loop. Therefore, to find a specific provider, simply search for it using a loop.

The Service-Based Keywords

Modules support services through the use of the keywords **provides**, **uses**, and **with**. Essentially, a module specifies that it provides a service with a **provides** statement. A module indicates that it requires a service with a **uses** statement. The specific type of service provider is declared by **with**. When used together, they enable you to specify a module that provides a service, a module that needs that service, and the specific implementation of that service. Furthermore, the module system ensures that the service and service providers are available and will be found.

Here is the general form of **provides**:

```
provides serviceType with implementationTypes;
```

Here, *serviceType* specifies the type of the service, which is often an interface, although abstract classes are also used. A comma-separated list of

the implementation types is specified by *implementationTypes*. Therefore, to provide a service, the module indicates both the name of the service and its implementation.

Here is the general form of the **uses** statement:

```
uses serviceType;
```

Here, *serviceType* specifies the type of the service required.

A Module-Based Service Example

To demonstrate the use of services, we will add a service to the modular application example that we have been evolving. For simplicity, we will begin with the first version of the application shown at the start of this chapter. To it we will add two new modules. The first is called **userfuncs**. It will define interfaces that support functions that perform binary operations in which each argument is an **int** and the result is an **int**. The second module is called **userfuncsimp**, and it contains concrete implementations of the interfaces.

Begin by creating the necessary source directories:

1. Under the **appsrc** directory, add directories called **userfuncs** and **userfuncsimp**.
2. Under **userfuncs**, add the subdirectory also called **userfuncs**. Under that directory, add the subdirectory **binaryfuncs**. Thus, beginning with **appsrc**, you will have created this tree:
`appsrc\userfuncs\userfuncs\binaryfuncs`
3. Under **userfuncsimp**, add the subdirectory also called **userfuncsimp**. Under that directory, add the subdirectory **binaryfuncsimp**. Thus, beginning with **appsrc**, you will have created this tree:
`appsrc\userfuncsimp\userfuncsimp\binaryfuncsimp`

This example expands the original version of the application by providing support for functions beyond those built into the application. Recall that the **SimpleMathFuncs** class supplies three built-in functions: **isFactor()**, **lcf()**, and **gcf()**. Although it would be possible to add more

functions to this class, doing so requires modifying and recompiling the application. By implementing services, it becomes possible to “plug in” new functions at run time, without modifying the application, and that is what this example will do. In this case, the service supplies functions that take two **int** arguments and return an **int** result. Of course, other types of functions can be supported if additional interfaces are provided, but support for binary integer functions is sufficient for our purposes and keeps the source code size of the example manageable.

The Service Interfaces

Two service-related interfaces are needed. One specifies the form of an action, and the other specifies the form of the provider of that action. Both go in the **binaryfuncs** directory, and both are in the **userfuncs.binaryfuncs** package. The first, called **BinaryFunc**, declares the form of a binary function. It is shown here:

```
// This interface defines a function that takes two int
// arguments and returns an int result. Thus, it can
// describe any binary operation on two ints that
// returns an int.

package userfuncs.binaryfuncs;

public interface BinaryFunc {
    // Obtain the name of the function.
    public String getName();

    // This is the function to perform. It will be
    // provided by specific implementations.
    public int func(int a, int b);
}
```

BinaryFunc declares the form of an object that can implement a binary integer function. This is specified by the **func()** method. The name of the function is obtainable from **getName()**. The name will be used to determine what type of function is implemented. This interface is implemented by a class that supplies a binary function.

The second interface declares the form of the service provider. It is called **BinFuncProvider** and is shown here:

```
// This interface defines the form of a service provider that
// obtains BinaryFunc instances.
package userfuncs.binaryfuncs;

import userfuncs.binaryfuncs.BinaryFunc;

public interface BinFuncProvider {
    // Obtain a BinaryFunc.
    public BinaryFunc get();
}
```

BinFuncProvider declares only one method, **get()**, which is used to obtain an instance of **BinaryFunc**. This interface must be implemented by a class that wants to provide instances of **BinaryFunc**.

The Implementation Classes

In this example, two concrete implementations of **BinaryFunc** are supported. The first is **AbsPlus**, which returns the sum of the absolute values of its arguments. The second is **AbsMinus**, which returns the result of subtracting the absolute value of the second argument from the absolute value of the first argument. These are provided by the classes **AbsPlusProvider** and **AbsMinusProvider**. The source code for these classes must be stored in the **binaryfuncsimp** directory, and they are all part of the **userfuncsimp.binaryfuncsimp** package.

The code for **AbsPlus** is shown here:

```
// AbsPlus provides a concrete implementation of
// BinaryFunc. It returns the result of abs(a) + abs(b).
package userfuncsimp.binaryfuncsimp;

import userfuncs.binaryfuncs.BinaryFunc;

public class AbsPlus implements BinaryFunc {

    // Return name of this function.
    public String getName() {
        return "absPlus";
    }

    // Implement the AbsPlus function.
    public int func(int a, int b) { return Math.abs(a) + Math.abs(b); }
}
```

AbsPlus implements **func()** such that it returns the result of adding the absolute values of **a** and **b**. Notice that **getName()** returns the "absPlus" string. It identifies this function.

The **AbsMinus** class is shown next:

```
// AbsMinus provides a concrete implementation of
// BinaryFunc. It returns the result of abs(a) - abs(b).

package userfuncsimp.binaryfuncsimp;

import userfuncs.binaryfuncs.BinaryFunc;

public class AbsMinus implements BinaryFunc {

    // Return name of this function.
    public String getName() {
        return "absMinus";
    }

    // Implement the AbsMinus function.
    public int func(int a, int b) { return Math.abs(a) - Math.abs(b); }
}
```

Here, **func()** is implemented to return the difference between the absolute values of **a** and **b**, and the string "absMinus" is returned by **getName()**.

To obtain an instance of **AbsPlus**, the **AbsPlusProvider** is used. It implements **BinFuncProvider** and is shown here:

```
// This is a provider for the AbsPlus function.

package userfuncsimp.binaryfuncsimp;

import userfuncs.binaryfuncs.*;

public class AbsPlusProvider implements BinFuncProvider {

    // Provide an AbsPlus object.
    public BinaryFunc get() { return new AbsPlus(); }
}
```

The **get()** method simply returns a new **AbsPlus()** object. Although this provider is very simple, it is important to point out that some service providers will be much more complex.

The provider for **AbsMinus** is called **AbsMinusProvider** and is shown next:

```
// This is a provider for the AbsMinus function.

package userfuncsimp.binaryfuncsimp;

import userfuncs.binaryfuncs.*;

public class AbsMinusProvider implements BinFuncProvider {

    // Provide an AbsMinus object.
    public BinaryFunc get() { return new AbsMinus(); }
}
```

Its **get()** method returns an object of **AbsMinus**.

The Module Definition Files

Next, two module definition files are needed. The first is for the **userfuncs** module. It is shown here:

```
module userfuncs {
    exports userfuncs.binaryfuncs;
}
```

This code must be contained in a **module-info.java** file that is in the **userfuncs** module directory. Notice that it exports the **userfuncs.binaryfuncs** package. This is the package that defines the **BinaryFunc** and **BinFuncProvider** interfaces.

The second **module-info.java** file is shown next. It defines the module that contains the implementations. It goes in the **userfuncsimp** module directory.

```
module userfuncsimp {
    requires userfuncs;

    provides userfuncs.binaryfuncs.BinFuncProvider with
        userfuncsimp.binaryfuncsimp.AbsPlusProvider,
        userfuncsimp.binaryfuncsimp.AbsMinusProvider;
}
```

This module requires **userfuncs** because that is where **BinaryFunc** and **BinFuncProvider** are contained, and those interfaces are needed by the implementations. The module provides **BinFuncProvider** implementations with the classes **AbsPlusProvider** and **AbsMinusProvider**.

Demonstrate the Service Providers in MyModAppDemo

To demonstrate the use of the services, the **main()** method of **MyModAppDemo** is expanded to use **AbsPlus** and **AbsMinus**. It does so by loading them at run time by use of **ServiceLoader.load()**. Here is the updated code:


```
// Now, use service-based, user-defined operations.

// Get a service loader for binary functions.
ServiceLoader<BinFuncProvider> ldr =
    ServiceLoader.load(BinFuncProvider.class);

BinaryFunc binOp = null;

// Find the provider for absPlus and obtain the function.
for(BinFuncProvider bfp : ldr) {
    if(bfp.get().getName().equals("absPlus")) {
        binOp = bfp.get();
        break;
    }
}

if(binOp != null)
    System.out.println("Result of absPlus function: " +
                       binOp.func(12, -4));
else
    System.out.println("absPlus function not found");

binOp = null;

// Now, find the provider for absMinus and obtain the function.
for(BinFuncProvider bfp : ldr) {
    if(bfp.get().getName().equals("absMinus")) {
        binOp = bfp.get();
        break;
    }
}

if(binOp != null)
    System.out.println("Result of absMinus function: " +
                       binOp.func(12, -4));
else
    System.out.println("absMinus function not found");

}
```

Let's take a close look at how a service is loaded and executed by the preceding code. First, a service loader for services of type **BinFuncProvider** is created with this statement:

```
ServiceLoader<BinFuncProvider> ldr =  
    ServiceLoader.load(BinFuncProvider.class);
```

Notice that the type parameter to **ServiceLoader** is **BinFuncProvider**. This is also the type used in the call to **load()**. This means that providers that implement this interface will be found. Thus, after this statement executes, **BinFuncProvider** classes in the module will be available through **ldr**. In this case, both **AbsPlusProvider** and **AbsMinusProvider** will be available.

Next, a reference of type **BinaryFunc** called **binOp** is declared and initialized to **null**. It will be used to refer to an implementation that supplies a specific type of binary function. Next, the following loop searches **ldr** for one that has the "absPlus" name.

```
// Find the provider for absPlus and obtain the function.  
for(BinFuncProvider bfp : ldr) {  
    if(bfp.getName().equals("absPlus")) {  
        binOp = bfp.get();  
        break;  
    }  
}
```

Here, a for-each loop iterates through **ldr**. Inside the loop, the name of the function supplied by the provider is checked. If it matches "absPlus", that function is assigned to **binOp** by calling the provider's **get()** method.

Finally, if the function is found, as it will be in this example, it is executed by this statement:

```
if(binOp != null)  
    System.out.println("Result of absPlus function: " +  
                      binOp.func(12, -4));
```

In this case, because **binOp** refers to an instance of **AbsPlus**, the call to **func()** performs an absolute value addition. A similar sequence is used to find and execute **AbsMinus**.

Because **MyModAppDemo** now uses **BinFuncProvider**, its module definition file must include a **uses** statement that specifies this fact. Recall that **MyModAppDemo** is in the **appstart** module. Therefore, you must change the **module-info.java** file for **appstart** as shown here:

```
// Module definition for the main application module.  
// It now uses BinFuncProvider.  
module appstart {  
    // Requires the modules appfuncs and userfuncs.  
    requires appfuncs;  
    requires userfuncs;  
  
    // appstart now uses BinFuncProvider.  
    uses userfuncs.binaryfuncs.BinFuncProvider;  
}
```

Compile and Run the Module-Based Service Example

Once you have performed all of the preceding steps, you can compile and run the example by executing the following commands:

```
javac -d appmodules --module-source-path appsrc  
    appsrc\userfuncsimp\module-info.java  
    appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java  
  
java --module-path appmodules -m appstart/appstart.mymodappdemo.MyModAppDemo
```

Here is the output:

```
2 is a factor of 10  
Smallest factor common to both 35 and 105 is 5  
Largest factor common to both 35 and 105 is 7  
Result of absPlus function: 16  
Result of absMinus function: 8
```

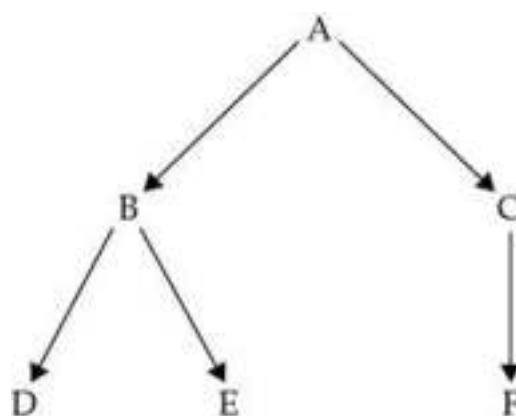
As the output shows, the binary functions were located and executed. It is important to emphasize that if either the **provides** statement in the **userfuncsimp** module or the **uses** statement in the **appstart** module were missing, the application would fail.

One last point: The preceding example was kept very simple in order to clearly illustrate module support for services, but much more sophisticated uses are possible. For example, you might use a service to provide a `sort()` method that sorts a file. Various sorting algorithms could be supported and made available through the service. The specific sort could then be chosen based on the desired run-time characteristics, the nature and/or size of the data, and whether random access to the data is supported. You might want to try implementing such a service as a way to further experiment with services in modules.

Module Graphs

A term you are likely to encounter when working with modules is *module graph*. During compilation, the compiler resolves the dependence relationships between modules by creating a module graph that represents the dependences. The process ensures that *all dependences* are resolved, including those that occur indirectly. For example, if module A requires module B, and B requires module C, then the module graph will contain module C even if A does not use it directly.

Module graphs can be depicted visually in a drawing to illustrate the relationship between modules. Here is a simple example. Assume six modules called **A**, **B**, **C**, **D**, **E**, and **F**. Further assume that **A** requires **B** and **C**, **B** requires **D** and **E**, and **C** requires **F**. The following visually depicts this relationship. (Because **java.base** is automatically included, it is not shown in the diagram.)



In Java, the arrows point from the dependent module to the required module. Thus, a drawing of a module graph depicts what modules have access to what other modules. Frankly, only the smallest applications can have their module graphs visually represented because of the complexity typically involved in many commercial applications.

Three Specialized Module Features

The preceding discussions have described the key features of modules supported by the Java language, and they are the features on which you will typically rely when creating your own modules. However, there are three additional module-related features that can be quite useful in certain circumstances. These are the **open** module, the **opens** statement, and the use of **requires static**. Each of these features is designed to handle a specialized situation, and each constitutes a fairly advanced aspect of the module system. That said, it is important for all Java programmers to have a general understanding of their purpose.

Open Modules

As you learned earlier in this chapter, by default, the types in a module's packages are accessible only if they are explicitly exported via an **exports** statement. While this is usually what you will want, there can be circumstances in which it is useful to enable run-time access to all packages in the module, whether a package is exported or not. To allow this, you can create an *open module*. An open module is declared by preceding the **module** keyword with the **open** modifier, as shown here:

```
open module moduleName {  
    // module definition  
}
```

In an open module, types in all its packages are accessible at run time. Understand, however, that only those packages that are explicitly exported are available at compile time. Thus, the **open** modifier affects only run-time accessibility. The primary reason for an open module is to enable the packages in the module to be accessed through reflection. As explained in

[Chapter 12](#), reflection is the feature that lets a program analyze code at run time.

The **opens** Statement

It is possible for a module to open a specific package for run-time access by other modules and for reflective access rather than opening an entire module. To do so, use the **opens** statement, shown here:

```
opens packageName;
```

Here, *packageName* specifies the package to open. It is also possible to include a **to** clause, which names those modules for which the package is opened.

It is important to understand **opens** does not grant compile-time access. It is used only to open a package for run-time and reflective access. However, you can both export and open a module. One other point: an **opens** statement cannot be used in an open module. Remember, all packages in an open module are already open.

requires static

As you know, **requires** specifies a dependence that, by default, is enforced both during compilation and at run time. However, it is possible to relax this requirement in such a way that a module is not required at run time. This is accomplished by use of the **static** modifier in a **requires** statement. For example, this specifies that **mymod** is required for compilation, but not at run time:

```
requires static mymod;
```

In this case, the addition of **static** makes **mymod** optional at run time. This can be helpful in a situation in which a program can utilize functionality if it is present, but not require it.

Introducing jlink and Module JAR Files

As the preceding discussions have shown, modules represent a substantial enhancement to the Java language. The module system also supports enhancements at run time. One of the most important is the ability to create a run-time image that is specifically tailored to your application. To accomplish this, you can use a JDK tool called **jlink**. It combines a group of modules into an optimized run-time image. You can use **jlink** to link modular JAR files, JMOD files, or even modules in their unarchived, “exploded directory” form.

Linking Files in an Exploded Directory

Let’s look first at using **jlink** to create a run-time image from unarchived modules. That is, the files are contained in their raw form in a fully expanded (i.e., exploded) directory. Assuming a Windows environment, the following command links the modules for the first example in this chapter. It must be executed from a directory *directly above mymodapp*.

```
jlink --launcher MyModApp=appstart/appstart.mymodappdemo.MyModAppDemo  
--module-path "%JAVA_HOME%\jmods;mymodapp\appmodules  
--add-modules appstart --output mylinkedmodapp
```

Let’s look closely at this command. First, the option **--launcher** tells **jlink** to create a command that starts the application. It specifies the name of the application and the path to the main class. In this case, the main class is **MyModAppDemo**. The **--module-path** option specifies the path to the required modules. The first is the path to the platform modules; the second is the path to the application modules. Notice the use of the environmental variable **JAVA_HOME**. It represents the path to the standard JDK directory. For example, in a standard Windows installation, the path will typically be something similar to "C:\program files"\java\jdk-17\jmods, but the use of **JAVA_HOME** is both shorter and able to work no matter in what directory the JDK was installed. The **--add-modules** option specifies the module or modules to add. Notice that only **appstart** is specified. This is because **jlink** automatically resolves all dependencies and includes all required modules. Finally, **--output** specifies the output directory.

After you run the preceding command, a directory called **mylinkedmodapp** will have been created that contains the run-time image. In its **bin** directory, you will find a launcher file called **MyModApp** that

you can use to run the application. For example, in Windows, this will be a batch file that executes the program.

Linking Modular JAR Files

Although linking modules from their exploded directory is convenient, when working on real-world code, you will often be using JAR files. (Recall that JAR stands for Java ARchive. It is a file format typically used for application deployment.) In the case of modular code, you will be using *modular JAR files*. A modular JAR file is one that contains a **module-info.class** file. Beginning with JDK 9, the **jar** tool has the ability to create modular JAR files. For example, it can now recognize a module path. Once you have created modular JAR files, you can use **jlink** to link them into a run-time image. To understand the process, let's work through an example. Again assuming the first example in this chapter, here are the **jar** commands that create modular JAR files for the **MyModAppDemo** program. Each must be executed from a directory directly above **mymodapp**. Also, you will need to create a directory called **applib** under **mymodapp**.

```
jar --create --file=mymodapp\applib\appfuncs.jar  
-C mymodapp\appmodules\appfuncs .  
  
jar --create --file=mymodapp\applib\appstart.jar  
--main-class=appstart.mymodappdemo.MyModAppDemo  
-C mymodapp\appmodules\appstart .
```

Here, **--create** tells **jar** to create a new JAR file. The **--file** option specifies the name of the JAR file. The files to include are specified by the **-C** option. The class that contains **main()** is specified by the **--main-class** option. After running these commands, the JAR files for the application will be in the **applib** directory under **mymodapp**.

Given the modular JAR files just created, here is the command that links them:

```
jlink --launcher MyModApp=appstart  
--module-path "%JAVA_HOME%\jmods;mymodapp\applib  
--add-modules appstart --output mylinkedmodapp
```

Here, the module path to the JAR files is specified, not the path to the exploded directories. Otherwise, the **jlink** command is the same as before.

As a point of interest, you can use the following command to run the application directly from the JAR files. It must be executed from a directory directly above **mymodapp**.

```
java -p mymodapp\applib -m appstart
```

Here, **-p** specifies the module path, and **-m** specifies the module that contains the program's entry point.

JMOD Files

The **jlink** tool can also link files that use the newer JMOD format introduced by JDK 9. JMOD files can include things that are not applicable to a JAR file. They are created by the new **jmod** tool. Although most applications will still use module JAR files, JMOD files will be of value in specialized situations. As a point of interest, beginning with JDK 9, the platform modules are contained in JMOD files.

NOTE **jlink** can also be used by the recently added **jpackage** tool. This tool can create a natively installable application.

A Brief Word About Layers and Automatic Modules

When learning about modules, you are likely to encounter reference to two additional module-related features. These are *layers* and *automatic modules*. Both are designed for specialized, advanced work with modules or when migrating preexisting applications. Although it is likely that most programmers will not need to make use of these features, a brief description of each is given here in the interest of completeness.

A module layer associates the modules in a module graph with a class loader. Thus, different layers can use different class loaders. Layers enable certain specialized types of applications to be more easily constructed.

An automatic module is created by specifying a nonmodular JAR file on the module path, with its name being automatically derived. (It is also

possible to explicitly specify a name for an automatic module in the manifest file.) Automatic modules enable normal modules to have a dependence on code in the automatic module. Automatic modules are provided as an aid in migration from pre-modular code to fully modular code. Thus, they are primarily a transitional feature.

Final Thoughts on Modules

The preceding discussions have introduced and demonstrated the core elements of Java's module system. These are the features about which every Java programmer should have at least a basic understanding. As you might guess, the module system provides additional features that give you fine-grained control over the creation and use of modules. For example, both **javac** and **java** have many more options related to modules than described in this chapter. Because modules are a significant addition to Java, it is likely that the module system will evolve over time. You will want to watch for enhancements to this innovative aspect of Java.

CHAPTER

17

Switch Expressions, Records, and Other Recently Added Features

A key attribute of Java has been its ability to adapt to the increasing demands of the modern computing environment. Over the years, Java has incorporated many new features, each responding to changes in the computing environment or to innovations in computer language design. This ongoing process has enabled Java to remain one of the world's most important and popular computer languages. As explained earlier, this book has been updated for JDK 21, which is a long-term support (LTS) version of Java. JDK 21 and its LTS predecessor JDK 17 have added a number of features to the Java language since the previous LTS version, JDK 11. A few of the smaller additions have been described in the preceding chapters. In this chapter, several major additions are examined. They are

- Records
- Patterns in **instanceof**
- Enhancements to **switch**
- Text blocks
- Sealed classes and interfaces

Here is a brief description of each. Supported by the new keyword **record**, records enable you to create a class that is specifically designed to hold a group of values. A second form of **instanceof** has been added that

uses a type pattern. With this form, you can specify a variable that receives an instance of the type being tested if `instanceof` succeeds. The **switch** has been enhanced in a number of ways, the most impacting of which is the **switch expression**. A **switch** expression enables a **switch** to produce a value. *Text blocks* allow a string literal to occupy more than a single line. It is now possible to specify a *sealed* class or interface. A sealed class can be inherited by only explicitly specified subclasses. A sealed interface can be implemented by only explicitly specified classes or extended by only explicitly specified interfaces. Thus, sealing a class or interface gives you fine-grained control over its inheritance and implementation.

Records

Beginning with JDK 16, Java supports a special-purpose class called a *record*. A record is designed to provide an efficient, easy-to-use way to hold a group of values. For example, you might use a record to hold a set of coordinates; bank account numbers and balances; the length, width, and height of a shipping container; and so on. Because it holds a group of values, a record is commonly referred to as an *aggregate* type. However, the record is more than simply a means of grouping data, because records also have some of the capabilities of a class. In addition, a record has unique features that simplify its declaration and streamline access to its values. As a result, records make it much easier to work with groups of data. Records are supported by the new context-sensitive keyword **record**.

One of the central motivations for records is the reduction of the effort required to create a class whose primary purpose is to organize two or more values into a single unit. Although it has always been possible to use **class** for this purpose, doing so can entail writing a number of lines of code for constructors, getter methods, and possibly (depending on use) overriding one or more of the methods inherited from **Object**. As you will see, by creating a data aggregate by using **record**, these elements are handled automatically for you, greatly simplifying your code. Another reason for the addition of records is to enable a program to clearly indicate that the intended purpose of a class is to hold a grouping of data, rather than act as a full-featured class. Because of these advantages, records are a much welcomed addition to Java.

Record Basics

As stated, a record is a narrowly focused, specialized class. It is declared by use of the **record** context-sensitive keyword. As such, **record** is a keyword only in the context of a **record** declaration. Otherwise, it is treated as a user-defined identifier with no special meaning. Thus, the addition of **record** does not impact or break existing code.

The general form of a basic **record** declaration is shown here:

```
record recordName(component-list) {  
    // optional body statements  
}
```

As the general form shows, a **record** declaration has significant differences from a **class** declaration. First, notice that the record name is immediately followed by a comma-separated list of parameter declarations called a *component list*. This list defines the data that the record will hold. Second, notice that the body is optional. This is made possible because the compiler will automatically provide the elements necessary to store the data; construct a record; create *getter methods* to access the data; and override **toString()**, **equals()**, and **hashCode()** inherited from **Object**. As a result, for many uses of a record, no body is required because the **record** declaration itself fully defines the record.

Here is an example of a simple **record** declaration:

```
record Employee(String name, int idNum) {}
```

Here, the record name is **Employee** and it has two components: the string **name** and the integer **idNum**. It specifies no statements in its body, so its body is empty. As the names imply, such a record could be used to store the name and ID number of an employee.

Given the **Employee** declaration just shown, a number of elements are automatically created. First, private final fields for **name** and **idNum** are declared as type **String** and **int**, respectively. Second, public read-only accessor methods (getter methods) that have the same names and types as the record components **name** and **idNum** are provided. Therefore, these getter methods are called **name()** and **idNum()**. In general, each record

component will have a corresponding private final field and a read-only public getter method automatically created by the compiler.

Another element created automatically by the compiler will be the record's *canonical constructor*. This constructor has a parameter list that contains the same elements, in the same order, as the component list in the record declaration. The values passed to the constructor are automatically assigned to the corresponding fields in the record. In a **record**, the canonical constructor takes the place of the default constructor used by a **class**.

A **record** is instantiated by use of **new**, just the way you create an instance of a **class**. For example, this creates a new **Employee** object, with the name "Doe, John" and ID number 1047:

```
Employee emp = new Employee("Doe, John", 1047);
```

After this declaration executes, the private fields **name** and **idNum** for **emp** will contain the values "Doe, John" and 1047, respectively. Therefore, you can use the following statement to display the information associated with **emp**:

```
System.out.println("The employee ID for " + emp.name() + " is " +  
    emp.idNum());
```

The resulting output is shown here:

```
The employee ID for Doe, John is 1047
```

A key point about a record is that its data is held in private final fields and only getter methods are provided. Thus, the data that a record holds is immutable. In other words, once you construct a record, its contents cannot be changed. However, if a record holds a reference to some object, you can make a change to that object, but you cannot change to what object the reference in the record refers. Thus, in Java terms, records are said to be *shallowly immutable*.

The following program puts the preceding discussion into action. Records are often used as elements in a list. For example, a business might maintain a list of **Employee** records to store an employee's name along with his or her corresponding ID number. The following program shows a

simple example of such usage. It creates a small array of **Employee** records. It then cycles through the array, displaying the contents of each record.

```
// A simple Record example.

// Declare an employee record. This automatically creates a
// record class with private, final fields called name and idNum,
// and with read-only accessors called name() and idNum().
record Employee(String name, int idNum) {}

class RecordDemo {
    public static void main(String[] args) {
        // Create an array of Employee records.
        Employee[] empList = new Employee[4];

        // Create a list of employees that uses the Employee record.
        // Notice how each record is constructed. The arguments
        // are automatically assigned to the name and idNum fields in
        // record that is being created.
        empList[0] = new Employee("Doe, John", 1047);
        empList[1] = new Employee("Jones, Robert", 1048);
        empList[2] = new Employee("Smith, Rachel", 1049);
        empList[3] = new Employee("Martin, Dave", 1050);

        // Use the record accessors to display names and IDs.
        for(Employee e: empList)
            System.out.println("The employee ID for " + e.name() + " is " +
                               e.idNum());
    }
}
```

The output is shown here:

```
The employee ID for Doe, John is 1047
The employee ID for Jones, Robert is 1048
The employee ID for Smith, Rachel is 1049
The employee ID for Martin, Dave is 1050
```

Before continuing, it is important to mention some key points related to records. First, a **record** cannot inherit another class. However, all records implicitly inherit **java.lang.Record**, which specifies abstract overrides of the **equals()**, **hashCode()**, and **toString()** methods declared by **Object**. Implicit implementations of these methods are automatically created, based on the record declaration. A **record** type cannot be extended. Thus, all **record** declarations are considered final. Although a **record** cannot extend another class, it can implement one or more interfaces. With the exception of **equals**, you cannot use the names of methods defined by **Object** as names for a **record**'s components. Aside from the fields associated with a **record**'s components, any other fields must be **static**. Finally, a **record** can be generic.

Create Record Constructors

Although you will often find that the automatically supplied canonical constructor is precisely what you want, you can also declare one or more of your own constructors. You can also define your own implementation of the canonical constructor. You might want to declare a **record** constructor for a number of reasons. For example, the constructor could check that a value is within a required range, verify that a value is in the proper format, ensure that an object implements optional functionality, or confirm that an argument is not **null**. For a **record**, there are two general types of constructors that you can explicitly create: canonical and non-canonical, and there are some differences between the two. The creation of each type is examined here, beginning with defining your own implementation of the canonical constructor.

Declare a Canonical Constructor

Although the canonical constructor has a specific, predefined form, there are two ways that you can code your own implementation. First, you can explicitly declare the full form of the canonical constructor. Second, you can use what is called a *compact record constructor*. Each approach is examined here, beginning with the full form.

To define your own implementation of a canonical constructor, simply do so as you would with any other constructor, specifying the record's name and its parameter list. It is important to emphasize that for the

canonical constructor, the types and parameter names must be the same as those specified by the **record** declaration. This is because the parameter names are linked to the automatically created fields and accessor methods defined by the **record** declaration. Thus, they must agree in both type and name. Furthermore, each component must be fully initialized upon completion of the constructor. The following restrictions also apply: the constructor must be at least as accessible as its **record** declaration. Thus, if the access modifier for the **record** is **public**, the constructor must also be specified **public**. A constructor cannot be generic, and it cannot include a **throws** clause (though it is permitted to throw an unchecked exception at run time). It also cannot invoke another constructor defined for the record.

Here is an example of the **Employee** record that explicitly defines the canonical constructor. It uses this constructor to remove any leading or trailing whitespace from a name. This ensures that names are stored in a consistent manner.

```
record Employee(String name, int idNum) {  
  
    // Use a canonical constructor to remove any leading and trailing spaces  
    // that might be in the name string. This ensures that names are stored  
    // in a consistent manner.  
    public Employee(String name, int idNum) {  
        // Remove any leading and trailing spaces.  
        this.name = name.trim();  
        this.idNum = idNum;  
    }  
}
```

In the constructor, leading and/or trailing whitespace is removed by a call to **trim()**. Defined by the **String** class, **trim()** deletes all leading and trailing whitespace from a string and returns the result. (If there are no leading or trailing spaces, the original string is returned unaltered.) The resulting string is assigned to the field **this.name**. Thus, no **Employee** record **name** will contain leading or trailing spaces. Next, the value of **idNum** is assigned to **this.idNum**. Because the identifiers **name** and **idNum** are the same for both fields corresponding to the **Employee** components and for the names used by the canonical constructor's parameters, the field names must be qualified by **this**.

Although there is certainly nothing wrong with creating a canonical constructor as just shown, there is often an easier way: through the use of a *compact constructor*. A compact record constructor is declared by specifying the name of the record, but without parameters. The compact constructor implicitly has parameters that are the same as the record's components, and its components are automatically assigned the values of the arguments passed to the constructor. Within the compact constructor you can, however, alter one or more of the arguments prior to their value being assigned to the record.

The following example converts the previous canonical constructor into its compact form:

```
// Use a compact canonical constructor to remove any leading and
// trailing spaces from the name string.
public Employee {
    // Remove any leading and trailing spaces.
    name = name.trim();
}
```

Here, the result of **trim()** is called on the **name** parameter (which is implicitly declared by the compact constructor) and the result is assigned back to the **name** parameter. At the end of the compact constructor, the value of **name** is automatically assigned to its corresponding field. The value of the implicit **idNum** parameter is also assigned to its corresponding field at the end of the constructor. Because the parameters are implicitly assigned to their corresponding fields when the constructor ends, there is no need to initialize the fields explicitly. Moreover, it would not be legal to do so.

Here is a reworked version of the previous program that demonstrates the compact canonical constructor:

```
// Use a compact record constructor.

// Declare an employee record.
record Employee(String name, int idNum) {

    // Use a compact canonical constructor to remove any leading and
    // trailing spaces from the name string.
    public Employee {
        // Remove any leading and trailing spaces.
        name = name.trim();
    }
}

class RecordDemo2 {
    public static void main(String[] args) {
        Employee[] empList = new Employee[4];

        // Here, the name has no leading or trailing spaces.
        empList[0] = new Employee("Doe, John", 1047);

        // The next three names have leading and/or trailing spaces.
        empList[1] = new Employee(" Jones, Robert", 1048);
        empList[2] = new Employee("Smith, Rachel ", 1049);
        empList[3] = new Employee(" Martin, Dave ", 1050);

        // Use the record accessors to display names and IDs.
        // Notice that all leading and/or trailing spaces have been
        // removed from the name component by the constructor.
        for(Employee e: empList)
            System.out.println("The employee ID for " + e.name() + " is " +
                               e.idNum());
    }
}
```

The output is shown here:

```
The employee ID for Doe, John is 1047
The employee ID for Jones, Robert is 1048
The employee ID for Smith, Rachel is 1049
The employee ID for Martin, Dave is 1050
```

As you can see, the names have been standardized with leading and trailing spaces removed. To prove to yourself that the call to `trim()` is necessary to achieve this result, simply remove the compact constructor, recompile, and run the program. The leading and trailing spaces will still be in the names.

Declare a Non-canonical Constructor

Although the canonical constructor will often be sufficient, you can declare other constructors. The key requirement is that any non-canonical constructor must first call another constructor in the record via `this`. The constructor invoked will often be the canonical constructor. Doing this ultimately ensures that all fields are assigned. Declaring a non-canonical constructor enables you to create special-case records. For example, you might use such a constructor to create a record in which one or more of the components is given a default placeholder value.

The following program declares a non-canonical constructor for `Employee` that initializes the name to a known value, but gives the `idNum` field the special value `pendingID` (which is `-1`) to indicate an ID value is not available when the record is created:

```
// Use a non-canonical constructor in a record.

// Declare an employee record that explicitly declares both
// a canonical and non-canonical constructor.
record Employee(String name, int idNum) {

    // Use a static field in a record.
    static int pendingID = -1;

    // Use a compact canonical constructor to remove any leading and
    // trailing spaces from the name string.
    public Employee {
        // Remove any leading and trailing spaces.
        name = name.trim();
    }

    // This is a non-canonical constructor. Notice that it is
    // not passed an ID number. Instead, it passes pendingID to the
    // canonical constructor to create the record.
    public Employee(String name) {
        this(name, pendingID);
    }
}

class RecordDemo3 {
    public static void main(String[] args) {
        Employee[] empList = new Employee[4];
```

```

// Create a list of employees that uses the Employee record.
empList[0] = new Employee("Doe, John", 1047);
empList[1] = new Employee("Jones, Robert", 1048);
empList[2] = new Employee("Smith, Rachel", 1049);

// Here, the ID number is pending.
empList[3] = new Employee("Martin, Dave");

// Display names and IDs.
for(Employee e: empList) {
    System.out.print("The employee ID for " + e.name() + " is ");
    if(e.idNum() == Employee.pendingID) System.out.println("Pending");
    else System.out.println(e.idNum());
}
}
}

```

Pay special attention to the way that the record for Martin, Dave is created by use of the non-canonical constructor. That constructor passes the **name** argument to the canonical constructor, but specifies the value **pendingID** as the **idNum** value. This enables a placeholder record to be created without having to specify an ID number. One other point: Notice that the value **pendingID** is declared as a **static** field in **Employee**. As explained earlier, instance fields are not allowed in a **record** declaration, but a **static** field is legal.

Notice that this version of **Employee** declares both a canonical constructor and a non-canonical constructor. This is perfectly valid. A record can define as many different constructors as its needs, as long as all adhere to the rules defined for **record**.

It is important to emphasize that records are immutable. As it relates to this example, it means that when an ID value for Martin, Dave is obtained, the old record must be replaced by a new record that contains the ID number. It is not possible to alter the record to update the ID. The immutability of records is a primary attribute.

Another Record Constructor Example

Before leaving the topic of record constructors, we will look at one more example. Because a **record** is used to aggregate data, a common use of a

record constructor is to verify the validity or applicability of an argument. For example, before constructing the record, the constructor may need to determine if a value is out of range, in an improper format, or otherwise unsuitable for use. If an invalid condition is found, the constructor could create a default or error instance. However, often it would be better for the constructor to throw an exception. This way, the user of the **record** would immediately be aware of the error and could take steps to correct it.

In the preceding **Employee** record examples, names have been specified using the common convention of *lastname, firstname*, such as Doe, John. However, there was no mechanism to verify or enforce that this format was being used. The following version of the compact canonical constructor provides a limited check that the name has the format *lastname, firstname*. It does so by confirming that there is one and only one comma in the name and that there is at least one character (other than space) before and after the comma. Although a far more thorough, careful verification would be needed by a real-world program, this minimal check is sufficient to serve as an example of the validation role a record constructor might play.

Here is a version of the **Employee** record in which the compact canonical constructor throws an exception if the **name** component does not meet the minimal criteria required for the *lastname, firstname* format:

```
// Use a compact canonical constructor to remove any leading
// and trailing spaces in the name component. Also perform
// a basic check that the required format of lastname, firstname
// is passed to the name parameter.
public Employee {
    // Remove any leading and trailing spaces.
    name = name.trim();

    // Perform a minimalist check that name follows the
    // lastname, firstname format.
    //
    // First, confirm that name contains only one comma.
    int i = name.indexOf(',') ; // look for comma separating names.
    int j = name.lastIndexOf(',');
    if(i != j) throw
        new IllegalArgumentException("Multiple commas found.");
    //
    // Next, confirm that a comma is present after
    // at least one leading character, and that at least one
    // character follows the comma.
    if(i < 1 | name.length() == i+1) throw
        new IllegalArgumentException("Required format: last, first");
}
```

When using this constructor, the following statement is still correct:

```
empList[0] = new Employee("Doe, John", 1047);
```

However, the following three are invalid and will result in an exception:

```
// No comma between last and first name.
empList[1] = new Employee("Jones Robert", 1048);

// Extra commas.
empList[1] = new Employee("Jones, ,Robert", 1048);

// Missing last name.
empList[1] = new Employee("", Robert", 1048);
```

As an aside, you might find it interesting to think of ways that you can improve the ability of the constructor to verify that the name uses the

proper format. For example, you might want to explore an approach that uses a regular expression. (See [Chapter 31](#).)

Create Record Getter Methods

Although it is seldom necessary, it is possible to create your own implementation of a getter method. When you declare the getter, the implicit version is no longer supplied. One possible reason you might want to declare your own getter is to throw an exception if some condition is not met. For example, if a record holds a filename and a URL, the getter for the filename might throw a **FileNotFoundException** if the file is not present at the URL. There is a very important requirement, however, that applies to creating your getters: they must adhere to the principle that a record is immutable. Thus, a getter that returns an altered value is semantically questionable (and should be avoided) even though such code would be syntactically correct.

If you do declare a getter implementation, there are a number of rules that apply. A getter must have the same return type and name as the component that it obtains. It must also be explicitly declared public. (Thus, default accessibility is not sufficient for a getter declaration in a **record**.) No **throws** clause is allowed in a getter declaration. Finally, a getter must be non-generic and non-static.

A better alternative to overriding a getter in cases in which you want to obtain a modified value of a component is to declare a separate method with its own name. For example, assuming the **Employee** record, you might want to obtain only the last name from the **name** component. Using the standard getter to do this would entail modifying the value obtained from the component. Doing this is a bad idea because it would violate the immutability aspect of the record. However, you could declare another method, called **lastName()**, that returns only the last name. The following program demonstrates this approach. It also uses the format-checking constructor from the previous section to ensure that names are stored as *lastname, firstname*.

```
// Use an instance method in a record.

// This version of Employee provides a method called lastName()
// that returns only the last name of the name component.
// It also includes the version of the compact constructor that
// checks for the conventional lastname, firstname format.
record Employee(String name, int idNum) {

    // Use a compact canonical constructor to remove any leading
    // and trailing spaces in the name component. Also perform
    // a basic check that the required format of lastname, firstname.
    // is passed to the name parameter.
    public Employee {
        // Remove any leading and trailing spaces.
        name = name.trim();

        // Perform a minimalist check that name follows the
        // lastname, firstname format.
        //
        // First, confirm that name contains only one comma.
        int i = name.indexOf(',');
        int j = name.lastIndexOf(',');
        if(i != j) throw
            new IllegalArgumentException("Multiple commas found.");

        // Next, confirm that a comma is present after
        // at least one leading character, and that at least one
        // character follows the comma.
        if(i < 1 | name.length() == i+1) throw
            new IllegalArgumentException("Required format: last, first");
    }
}
```

```

// An instance method that returns only the last name
// without the first name.
String lastName() {
    return name.substring(0, name.trim().indexOf(','));
}
}

class RecordDemo4 {
    public static void main(String[] args) {
        Employee emp = new Employee("Jones, Robert", 1048);

        // Display the name component unmodified.
        System.out.println("Employee full name is " + emp.name());

        // Display only last name.
        System.out.println("Employee last name is " + emp.lastName());
    }
}

```

The output is shown here:

```

Employee full name is Jones, Robert
Employee last name is Jones

```

As the output shows, the implicit getter for the **name** component returns the name unaltered. The instance method **lastName()** obtains only the last name. With this approach, the immutable attribute of the **Employee** record is preserved, while still providing a convenient means of obtaining the last name.

Pattern Matching with instanceof

The traditional form of the **instanceof** operator was introduced in [Chapter 13](#). As explained there, **instanceof** evaluates to **true** if an object is of a specified type, or can be cast to that type. Beginning with JDK 16, a second form of the **instanceof** operator has been added to Java that supports the new *pattern matching* feature. In general terms, *pattern matching* defines a mechanism that determines if a value fits a general form. As it relates to **instanceof**, pattern matching is used to test the type of a value (which must

be a reference type) against a specified type. If the pattern matches, a *pattern variable* will receive a reference to the object matched by the pattern. Beginning with JDK 21 there are two kinds of patterns: *type patterns* and *record patterns*. Type patterns are described here. Record patterns are discussed later in this chapter.

The pattern matching form of **instanceof** is shown here:

objref instanceof type pattern-var

If **instanceof** succeeds, *pattern-var* will be created and contain a reference to the object that matches the pattern. If it fails, *pattern-var* is never created. This form of **instanceof** succeeds if the object referred to by *objref* can be cast to *type* and the static type of *objref* is not a subtype of *type*.

For example, the following fragment creates a **Number** reference called **myOb** that refers to an **Integer** object. (Recall that **Number** is a superclass of all numeric primitive-type wrappers.) It then uses the **instanceof** operator to confirm that the object referred to by **myOb** is an **Integer**, which it will be in this example. This results in an object called **iObj** of type **Integer** being instantiated that contains the matched value.

```
Number myOb = Integer.valueOf(9);

// Use the pattern matching version of instanceof.
if(myOb instanceof Integer iObj) {
    // iObj is known and in scope here.
    System.out.println("iObj refers to an integer: " + iObj);
}
// iObj does not exist here
```

As the comments indicate, **iObj** is known only within the scope of the **if** clause. It is not known outside of the **if**. It also would not be known within an **else** clause, should one have been included. It is crucial to understand that the pattern variable **iObj** is created only if the pattern matching succeeds.

The primary advantage of the pattern matching form of **instanceof** is that it reduces the amount of code that was typically needed by the traditional form of **instanceof**. For example, consider this functionally

equivalent version of the preceding example that uses the traditional approach:

```
// Use a traditional instanceof.  
if(myOb instanceof Integer) {  
    // Use an explicit cast to obtain iObj.  
    Integer iObj = (Integer) myOb;  
    System.out.println("iObj refers to an integer: " + iObj);  
}
```

With the traditional form, a separate declaration statement and explicit cast are required to create the **iObj** variable. The pattern matching form of **instanceof** streamlines the process.

Pattern Variables in a Logical AND Expression

An **instanceof** can be used in a logical AND expression. However, you need to remember that the pattern variable is only in scope after it has been created. For example, the following **if** succeeds only when **myOb** refers to an **Integer** and its value is nonnegative. Pay special attention to the expression in the **if**:

```
if((myOb instanceof Integer iObj) && (iObj >= 0)) { // is OK  
    // myOb is both an Integer and nonnegative.  
    // ...  
}
```

The **iObj** pattern variable is created only if the left side of the **&&** (the part that contains the **instanceof** operator) is true. Notice that **iObj** is also used by the right side. This is possible because the short-circuit form of the AND logical operator is used, and the right side is evaluated *only if* the left succeeds. Thus, if the right-side operand is evaluated, **iObj** will be in scope. However, if you tried to write the preceding statement using the **&** operator like this:

```
if((myOb instanceof Integer iObj) & (iObj >= 0)) { // Error!  
    // myOb is both an Integer and nonnegative.  
    // ...  
}
```

a compilation error would occur because **iObj** will not be in scope if the left side fails. Recall that the **&** operator causes both sides of the expression to be evaluated, but **iObj** is only in scope if the left side is true. This error is caught by the compiler. A related situation occurs with this fragment:

```
int count = 10;
if((count < 100) && myOb instanceof Integer iObj) { // is OK
    // myOb is both an Integer and nonnegative, and count is less than 100.

    iObj = count;
    // ...
}
```

This fragment compiles because the **if** block will execute only when both sides of the **&&** are true. Thus, the use of **iObj** in the **if** block is valid. However, a compilation error will result if you tried to use the **&** rather than the **&&**, as shown here:

```
if((count < 100) & myOb instanceof Integer iObj) { // Error!
```

In this case, the compiler cannot know whether or not **iObj** will be in scope in the **if** block because the right side of the **&** will not necessarily be evaluated.

One other point: A logical expression cannot introduce the same pattern variable more than once. For example, in a logical AND, it is an error if both operands create the same pattern variable.

Pattern Matching in Other Statements

Although a frequent use of the pattern matching form of **instanceof** is in an **if** statement, it is by no means limited to that use. It can also be employed in the conditional portion of the loop statements. As an example, imagine that you are processing a collection of objects, perhaps contained in an array. Furthermore, at the start of the array are several strings, and you want to process those strings, but not any of the remaining objects in the list. The following sequence accomplishes this task with a **for** loop in which the condition uses **instanceof** to confirm that an object in the array is a **String** and to obtain that string for processing within the loop. Thus, pattern

matching is used to control the execution of a **for** loop and to obtain the next value for processing.

```
Object[] someObjs = {
    new String("Alpha"),
    new String("Beta"),
    new String("Omega"),
    Integer.valueOf(10)
};

int i;

// This loop iterates until an element is not a String, or the end
// of the array is reached.
for(i = 0; (someObjs[i] instanceof String str) && (i < someObjs.length); i++) {
    System.out.println("Processing " + str);
    // ...
}

System.out.println("The first " + i + " entries in the list are strings.");
```

The output from this fragment is shown here:

```
Processing Alpha
Processing Beta
Processing Omega
The first 3 entries in the list are strings.
```

The pattern matching form of **instanceof** can also be useful in a **while** loop. For example, here is the preceding **for** loop, recoded as a **while**:

```
i = 0;
while((someObjs[i] instanceof String str) && (i < someObjs.length)) {
    System.out.println("Processing " + str);
    i++;
}
```

Although it is technically possible to use the pattern matching **instanceof** in the conditional portion of a **do** loop, such use is severely

limited because the pattern variable cannot be used in body of the loop because it will not be in scope until the `instanceof` operator is executed.

Enhancements to switch

The `switch` statement has been part of Java since the start. It is a crucial element of Java's program control statements and provides for a multiway branch. Moreover, `switch` is so fundamental to programming that it is found in one form or another in other popular programming languages. The traditional form of `switch` was described in [Chapter 5](#). This is the form of `switch` that has always been part of Java. Beginning with JDK 14, `switch` has been substantially enhanced with the addition of four new features, and in JDK 21 a fifth, shown here:

- The `switch` expression
- The `yield` statement
- Support for a list of `case` constants
- The `case` with an arrow
- Pattern matching in `switch`

Each new feature is examined in detail in the discussions that follow, but here is a brief description: The `switch` expression is, essentially, a `switch` that produces a value. Thus, a `switch` expression can be used on the right side of an assignment, for example. The `yield` statement specifies a value that is produced by a `switch` expression. It is now possible to have more than one `case` constant in a `case` statement through the use of a list of constants. A second form of `case` has been added that uses an arrow (`->`) instead of a colon. The arrow gives `case` new capabilities.

Collectively, the enhancements to `switch` represent a fairly significant change to the Java language. Not only do they provide new capabilities, but in some situations, they also offer superior alternatives to traditional approaches. Because of this, a solid understanding of the “how” and “why” behind the new `switch` features is important.

One of the best ways to understand the `switch` enhancements is to start with an example that uses a traditional `switch` and then gradually incorporate each new feature. This way, the use and benefit of the enhancements will be clearly apparent. To begin, imagine some device that

produces integer codes that indicate various events, and you want to associate a priority level with each event code. Most events will have a normal priority, but a few will have a higher priority. Here is a program that uses a traditional **switch** statement to supply a priority level given an event code:

```
// Use a traditional switch to set a priority level based on which
// event code is matched.
class TraditionalSwitch {

    public static void main(String[] args) {
        int priorityLevel;

        int eventCode = 6010;

        // A traditional switch that supplies a value associated
        // with a case.
        switch(eventCode) {
            case 1000: // In this traditional switch, case stacking is used.
            case 1205:
            case 8900:
                priorityLevel = 1;
                break;
            case 2000:
            case 6010:
            case 9128:
                priorityLevel = 2;
                break;
            case 1002:
            case 7023:
            case 9300:
                priorityLevel = 3;
                break;
            default: // normal priority
                priorityLevel = 0;
        }

        System.out.println("Priority level for event code " + eventCode +
                           " is " + priorityLevel);
    }
}
```

The output is shown here:

```
Priority level for event code 6010 is 2
```

There is certainly nothing wrong with using a traditional **switch** as shown in the program, and this is the way Java code has been written for more than two decades. However, as the following sections will show, in many cases, the traditional **switch** can be improved by use of the enhanced **switch** features.

Use a List of **case** Constants

We begin with one of the easiest ways to modernize a traditional **switch**: by use of a list of **case** constants. In the past, when two constants were both handled by the same code sequence, *case stacking* was employed, and this is the approach used by the preceding program. For example, here are how the **cases** for 1000, 1205, and 8900 are handled:

```
case 1000: // In this traditional switch, case stacking is used.  
case 1205:  
case 8900:  
    priorityLevel = 1;  
    break;
```

The stacking of **case** statements enables all three **case** statements to use the same code sequence to set **priorityLevel** to 1. As explained in [Chapter 5](#), in a traditional-style **switch**, the stacking of **cases** is made possible because execution falls through each **case** until a **break** is encountered. Although this approach works, a more elegant solution can be achieved by use of a *case constant list*.

Beginning with JDK 14, you can specify more than one **case** constant in a single **case**. To do so, simply separate each constant with a comma. For example, here is a more compact way to code the **case** for 1000, 1205, and 8900:

```
case 1000, 1205, 8900: // use a case list  
    priorityLevel = 1;  
    break;
```

Here is the entire **switch**, rewritten to use lists of **case** constants:

```
// This switch uses a list of constants with each case.
switch(eventCode) {
    case 1000, 1205, 8900:
        priorityLevel = 1;
        break;
    case 2000, 6010, 9128:
        priorityLevel = 2;
        break;
    case 1002, 7023, 9300:
        priorityLevel = 3;
        break;
    default: // normal priority
        priorityLevel = 0;
}
```

As you can see, the number of **case** statements has been reduced by six, making the **switch** easier to read and a bit more manageable. Although support for a **case** constant list does not by itself add any fundamentally new functionality to the **switch**, it does help streamline your code. In many situations, it also offers an easy way to improve existing code—especially when extensive **case**-stacking was previously employed. Thus, it is a feature that you can put to work immediately, with minimal code rewriting.

Introducing the **switch** Expression and the **yield** Statement

Of the enhancements to **switch**, the one that will have the most profound impact is the **switch expression**. A **switch** expression is, essentially, a **switch** that returns a value. Thus, it has all of the capabilities of a traditional **switch** statement, plus the ability to produce a result. This added capability makes the **switch** expression one of the more important additions to Java in recent years.

One way to supply the value of a **switch** expression is with the new **yield** statement. It has this general form:

```
yield value;
```

Here, *value* is the value produced by the **switch**, and it can be any expression compatible with the type of value required. A key point to understand about **yield** is that it immediately terminates the **switch**. Thus, it works somewhat like **break**, with the added capability of supplying a value. It is important to point out that **yield** is a context-sensitive keyword. This means that outside its use in a **switch** expression, **yield** is simply an identifier with no special meaning. However, if you use a method called **yield()**, it must be qualified. For example, if **yield()** is a non-**static** method within its class, you must use **this.yield()**.

It is very easy to specify a **switch** expression. Simply use the **switch** in a context in which a value is required, such as on the right side of an assignment statement, an argument to a method, or a return value. For example, this line indicates that a **switch** expression is being employed:

```
int x = switch(y) { // ...
```

Here, the **switch** result is being assigned to the **x** variable. A key point about using a **switch** expression is that each **case** (plus **default**) must produce a value (unless it throws an exception). In other words, each path through a **switch** expression must produce a result.

The addition of the **switch** expression simplifies the coding of situations in which each **case** sets the value of some variable. Such situations can occur in a number of different ways. For example, each **case** might set a **boolean** variable that indicates the success or failure of some action taken by the **switch**. Often, however, setting a variable is the *primary purpose* of the **switch**, as is the case with the **switch** used by the preceding program. Its job is to produce the priority level associated with an event code. With a traditional **switch** statement, each **case** statement must individually assign a value to the variable, and this variable becomes the de facto result of the **switch**. This is the approach used by the preceding programs, in which the value of the variable **priorityLevel** is set by each **case**. Although this approach has been used in Java programs for decades, the **switch** expression offers a better solution because the desired value is produced by the **switch** itself.

The following version of the program puts the preceding discussion into action by changing the **switch** statement into a **switch** expression:

```

// Convert a switch statement into a switch expression.
class SwitchExpr {

    public static void main(String[] args) {
        int eventCode = 6010;

        // This is a switch expression. Notice how its value is assigned
        // to the priorityLevel variable. Also notice how the value of the
        // switch is supplied by the yield statement.
        int priorityLevel = switch(eventCode) {
            case 1000, 1205, 8900:
                yield 1;
            case 2000, 6010, 9128:
                yield 2;
            case 1002, 7023, 9300:
                yield 3;
            default: // normal priority
                yield 0;
        };

        System.out.println("Priority level for event code " + eventCode +
                           " is " + priorityLevel);
    }
}

```

Look closely at the **switch** in the program. Notice that it differs in important ways from the one used in the previous examples. Instead of each **case** assigning a value to **priorityLevel** individually, this version assigns the outcome of the **switch** itself to the **priorityLevel** variable. Thus, only one assignment to **priorityLevel** is required, and the length of the **switch** is reduced. Using a **switch** expression also ensures that each **case** yields a value, thus avoiding the possibility of forgetting to give **priorityLevel** a value in one of the **cases**. Notice that the value of the **switch** is produced by the **yield** statement inside each **case**. As explained, **yield** causes immediate termination of the **switch**, so no fall-through from **case** to **case** will occur. Thus, no **break** statement is required, or allowed. One other thing to notice is the semicolon after the closing brace of the **switch**. Because this **switch** is used in an assignment, it must be terminated by a semicolon.

There is an important restriction that applies to a **switch** expression: the **case** statements must handle all of the values that might occur. Thus, a

switch expression must be *exhaustive*. For example, if its controlling expression is of type **int**, then all **int** values must be handled by the **switch**. This would, of course, constitute a very large number of **case** statements! For this reason, most **switch** expressions will have a **default** statement. The exception to this rule is when an enumeration is used, and each value of the enumeration is matched by a **case**.

Introducing the Arrow in a case Statement

Although the use of **yield** in the preceding program is a perfectly valid way to specify a value for a **switch** expression, it is not the only way to do so. In many situations, an easier way to supply a value is through the use of a new form of the **case** that substitutes **->** for the colon in a **case**. For example, this line:

```
case 'X': // ...
```

can be rewritten using the arrow like this:

```
case 'X' -> // ...
```

To avoid confusion, in this discussion we will refer to a **case** with an arrow as an *arrow case* and the traditional, colon-based form as a *colon case*. Although both forms will match the character X, the precise action of each style of **case** statement differs in three very important ways.

First, one arrow **case** *does not* fall through to the next **case**. Thus, there is no need to use **break**. Execution simply terminates at the end of an arrow **case**. Although the fall-through nature of a traditional colon-based **case** has always been part of Java, fall-through has been criticized because it can be a source for bugs, such as when the programmer forgets to add a **break** statement to prevent fall-through when fall-through is not desired. The arrow **case** avoids this situation. Second, the arrow **case** provides a “shorthand” way to supply a value when used in a **switch** expression. For this reason, the arrow **case** is often used in **switch** expressions. Third, the target of an arrow **case** must be either an expression, a block, or throw an exception. It cannot be a statement sequence, as is allowed with a traditional **case**. Thus, the arrow **case** will have one of these general forms:

```
case constant -> expression;  
case constant -> { block-of-statements }  
case constant -> throw ...
```

Of course, the first two forms represent the primary uses.

Arguably, the most common use of an arrow **case** is in a **switch** expression, and the most common target of the arrow **case** is an expression. Thus, it is here that we will begin. When the target of an arrow **case** is an expression, the value of that expression becomes the value of the **switch** expression when that **case** is matched. Thus, it provides a very efficient alternative to the **yield** statement in many situations. For example, here is the first **case** in the event code example rewritten to use an arrow **case**:

```
case 1000, 1205, 8900 -> 1;
```

Here, the value of the expression (which is 1) automatically becomes the value produced by the **switch** when this **case** is matched. In other words, the expression becomes the value yielded by the **switch**. Notice that this statement is quite compact, yet clearly expresses the intent to supply a value.

In the following program, the entire **switch** expression has been completely rewritten to use the arrow **case**:

```
// Use the arrow "shorthand" to supply the priority level.  
class SwitchExpr2 {  
  
    public static void main(String[] args) {  
        int eventCode = 6010;
```

```

// In this switch expression, notice how the value is supplied
// by use of an arrow case. Notice that no break statement is
// required (or allowed) to prevent fall-through.
int priorityLevel = switch(eventCode) {
    case 1000, 1205, 8900 -> 1;
    case 2000, 6010, 9128 -> 2;
    case 1002, 7023, 9300 -> 3;
    default -> 0; // normal priority
};

System.out.println("Priority level for event code " + eventCode +
    " is " + priorityLevel);
}
}

```

This produces the same output as before. Looking at the code, it is easy to see why this form of the arrow **case** is so appropriate for many types of **switch** expressions. It is compact and eliminates the need for a separate **yield** statement. Because the arrow **case** does not fall through, there is no need for a **break** statement. Each **case** terminates by yielding the value of its expression. Furthermore, if you compare this final version of the **switch** to the original, traditional **switch** shown at the start of this discussion, it is readily apparent how streamlined and expressive this version is. In combination, the **switch** enhancements offer a truly impressive way to improve the clarity and resiliency of your code.

A Closer Look at the Arrow case

The arrow **case** provides considerable flexibility. First, when using its expression form, the expression can be of any type. For example, the following is a valid **case** statement:

```
case -1 -> getErrorCode();
```

Here, the result of the call to **getErrorCode()** becomes the value of the enclosing **switch** expression. Here is another example:

```
case 0 -> normalCompletion = true;
```

In this case, the result of the assignment, which is **true**, becomes the value yielded. The key point is that any valid expression can be used as the target of the arrow **case** as long as it is compatible with the type required by the **switch**.

As mentioned, the target of the `->` can also be a block of code. You will need to use a block as the target of an arrow **case** whenever you need more than a single expression. For example, each **case** in this version of the event code program sets the value of a **boolean** variable called **stopNow** to indicate if immediate termination is required and then yields the priority level:

```
// Use blocks with an arrow.  
class BlockArrowCase {  
  
    public static void main(String[] args) {  
        boolean stopNow;
```

```
int eventCode = 9300;

// Use code blocks with an arrow. Again, notice
// that no break statement is required (or allowed)
// to prevent fall through. Because the target of an
// arrow is a block, yield must be used to supply a value.
int priorityLevel = switch(eventCode) {
    case 1000, 1205, 8900 -> { // use a block with an arrow
        stopNow = false;
        System.out.println("Alert");
        yield 1;
    }
    case 2000, 6010, 9128 -> {
        stopNow = false;
        System.out.println("Warning");
        yield 2;
    }
    case 1002, 7023, 9300 -> {
        stopNow = true;
        System.out.println("Danger");
        yield 3;
    }
    default -> {
        stopNow = false;
        System.out.println("Normal.");
        yield 0;
    }
};

System.out.println("Priority level for event code " + eventCode +
                   " is " + priorityLevel);
if(stopNow) System.out.println("Stop required.");
}
```

Here is the output:

```
Danger
Priority level for event code 9300 is 3
Stop required.
```

As this example shows, when using a block, you must use **yield** to supply a value to a **switch** expression. Furthermore, even though block targets are used, each path through the **switch** expression must still provide a value.

Although the preceding program provides a simple illustration of a block target of an arrow **case**, it also raises an interesting question. Notice that each **case** in the **switch** sets the value of two variables. The first is **priorityLevel**, which is the value yielded. The second is **stopNow**. Is there a way for a **switch** expression to yield more than one value? In a direct sense, the answer is “no” because only one value can be produced by the **switch**. However, it is possible to encapsulate two or more values within a class and yield an object of that class. Beginning with JDK 16, Java provides an especially streamlined and efficient way to accomplish this: the **record**. Described later in this chapter, a **record** aggregates two or more values into a single logical unit. As it relates to this example, a **record** could hold both the **priorityLevel** and the **stopNow** values, and this **record** could be yielded by the **switch** as a unit. Thus, a **record** offers a convenient way for a **switch** to yield more than a single value.

Although the arrow **case** is very helpful in a **switch** expression, it is important to emphasize that it is not limited to that use. The arrow **case** can also be used in a **switch** statement, which enables you to write **switches** in which no **case** fall-through can occur. In this situation, no **yield** statement is required (or allowed), and no value is produced by the **switch**. In essence, it works much like a traditional **switch** but without the fall-through. Here is an example:

```
// Use case arrows with a switch statement
class StatementSwitchWithArrows {

    public static void main(String[] args) {
        int up = 0;
        int down = 0;
        int left = 0;
        int right = 0;

        char direction = 'R';

        // Use arrows with a switch statement. Notice that
        // no value is produced.
        switch(direction) {
            case 'L' -> {
                System.out.println("Turning Left");
                left++;
            }
            case 'R' -> {
                System.out.println("Turning Right");
                right++;
            }
            case 'U' -> {
                System.out.println("Moving Up");
                up++;
            }
            case 'D' -> {
                System.out.println("Moving Down");
                down++;
            }
        }

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

In this program, the **switch** is a statement, not an expression. This is because of two reasons. First, no value is produced. Second, it is not exhaustive because no **default** statement is included. (Recall that **switch** expressions must be exhaustive, but not **switch** statements.) Notice, however, that because no fall-through occurs with an arrow **case**, no **break**

statement is needed. As a point of interest, because each **case** increases the value of a different variable, it would not be possible to transform this **switch** into an expression. What value would it produce? All four **cases** increment a different variable.

One last point: you cannot mix arrow **cases** with traditional colon **cases** in the same **switch**. You must choose one or the other. For example, this sequence is invalid:

```
// This won't work! You cannot mix a colon case with an arrow case.
switch(direction) {
    case 'L' -> {
        System.out.println("Turning Left");
        left++;
    }
    case 'R' : // Wrong! Can't mix case styles.
        System.out.println("Turning Right");
        right++;
        break;
    case 'U' -> {
        System.out.println("Moving Up");
        up++;
    }
    case 'D' -> {
        System.out.println("Moving Down");
        down++;
    }
}
```

Another switch Expression Example

To continue this overview of the **switch** enhancements, another example is presented. It uses a **switch** expression to determine whether a letter is an English-language vowel. It makes use of all of the new **switch** features. Pay special attention to the way Y is handled. In English, Y can be a vowel or a consonant. The program lets you specify which way you want the Y interpreted by the way the **yIsVowel** variable is set. To handle this special case, a block is used as the target of the **->**.

```

// Use a switch expression to determine if a character is an English vowel.
// Notice the use of a block as the target of an arrow case for Y.

class Vowels {

    public static void main(String[] args) {

        // If Y is to be counted as a vowel, set this
        // variable to true.
        boolean yIsVowel = true;

        char ch = 'Y';

        boolean isVowel = switch(ch) {
            case 'a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U' -> true;
            case 'y', 'Y' -> { if(yIsVowel) yield true; else yield false; }
            default -> false;
        };

        if(isVowel) System.out.println(ch + " is a vowel.");
    }
}

```

As an experiment, try rewriting this program using a traditional **switch**. As you will find, doing so results in a much longer, less manageable version. The new **switch** enhancements often provide a superior approach.

Pattern Matching in switch

JDK 21 introduces a convenient way to use pattern matching with **switch**, and it's the last of the enhancements to **switch**. It follows a similar syntax of pattern matching that you saw earlier in the context of the **instanceof** keyword, but within **switch** statements.

By now you are familiar with the power of **switch** statements to simplify code that chooses to perform different functions based on different values of a variable, but what about when the variable can be of a number of different types, enums, null values, and even ranges of values?

Take a look at the following code, which prints a message based on an object that identifies an employee of some kind at a company. The object

can come in many shapes and sizes: it can be a Java record representing an employee with name and ID, an implementation of an interface called Employee, a particular implementation of that interface, a Java enum representing the chief executive and chief financial officers of a company, an integer, a string, or something else entirely. Depending on its nature, you want the code to handle the object representing an employee differently. You might naturally think of using **instanceof** checks in a chain of if-else statements to separate out all the different cases. If you did, you would end up with something that looks like this:

```
interface Employee {};
static class RegularEmployee implements Employee {};
enum CSuiteEmployee implements Employee {CEO, CFO}
record EmpId(String name, int id) {}

static void handleId(Object empObject) {
    if (empObject == null) {
        System.out.println("Oops, the Id was null !");
    } else if (empObject instanceof Integer idInt) {
        if (idInt == -1 || idInt == 0) {
            System.out.println("A reserved employee Id");
        } else if (idInt < 0) {
            System.out.println("A negative employee Id: " + idInt);
        } else {
            System.out.println("A positive employee Id: " + idInt);
        }
    } else if (empObject instanceof String s) {
        System.out.println("A String Id: " + s);
    } else if (empObject instanceof EmpId eid) {
        System.out.println("An Employee Id: [" + eid.id + ", "
                           + eid.name + "]");
    } else if (empObject instanceof Employee e) {
        if (e instanceof CSuiteEmployee cse) {
            switch (cse) {
                case CEO -> System.out.println("It's the boss !");
                case CFO -> System.out.println("There's the money !");
            }
        }
    }
}
```

```

        }
    } else if (e instanceof RegularEmployee) {
        System.out.println("A regular employee.");
    } else {
        System.out.println("An irregular employee.");
    }
} else {
    System.out.println("Some other kind of employee: "
                       + empObject.toString());
}
}

```

In this example, you can see that the message the code formulates to print out depends upon the type of the employee object. You can also see in the case when the **id** is an **Integer**, the message printed is different for numerical values that are exactly equal to **-1** or **0**, or are negative or positive. The logic of this code to sift through all the possible types of employee is buried in the structure of the **if-else** statements, making it quite difficult to see how the different types and values are treated without deep study. Fortunately, by using pattern matching in **switch** statements, you can simplify this code by using the four new features of **switch**, described as follows.

You can use null in a case statement.

Before JDK 21, if you passed a null value into a **switch** statement, it would throw a **NullPointerException** at run time. This meant that you would need to have an explicit **null** check somewhere before your **switch** statement. But the new enhanced **switch** statements can have a **case** statement to deal with **null** values in the **switch** variable, like this:

```
case null -> // handle null value here
```

Note that if you don't provide a **case** statement to handle a **null** case, the **switch** statement still throws a **NullPointerException** in the case of a **null** value, just as before JDK 21.

You can use pattern matching within case statements.

This second expansion of the power of the **case** statement allows you to concisely check that the **switch** variable is of a given type and also assign a variable to it that you can use for processing this case. For example:

```
case String s -> System.out.println("It's a String with value:  
" + s);
```

You can refine the case statement to deal with specific values.

This is an important addition to the use of the contextual keyword **when** to the syntax because it allows you to express a **case** label that is conditional on the value of an object reference instead of the fixed values of a literal. For example, you can express a case label for a **String** value that only applies if the **String** has a certain form:

```
case String s when s.startsWith("A") -> System.out.println("It's a String"  
+ "beginning with the letter A");
```

You can use Java records and more easily use enum constants in switch statements.

Prior to JDK 21 you could only use enum constants in **switch** statements where the switch variable was that enum type. This limitation has been relaxed in JDK 21 to allow you to use enum constants in **case** statements when the switch variable is a more general type. You can also use a Java record in a **case** statement and in pattern matching form.

Let's combine all four of these new features to simplify the code you started out with and make it much more concise and readable:

```

// use patterns in switch to simplify processing
// the switch variable, here an object representing
// an id, depending on its type and value.
static void handleId(Object empObject) {
    switch (empObject) {
        case null -> System.out.println("Oops, the Id was null !");
        case Integer i -> {
            switch (i) {
                case -1, 0 -> System.out.println("A reserved employee Id");
                case Integer idInt
                    when idInt < 0 -> System.out.println("A negative employee Id: "
                        + idInt);
                case Integer idInt -> System.out.println("A positive employee Id: "
                        + idInt);
            }
        }
        case String s -> System.out.println("A String Id " + s);
        case EmpId eid -> System.out.println("An Employee Id: [" + eid.id() + ", "
            + eid.name + "]");
        case Employee e -> {
            switch (e) {
                case CSuiteEmployee.CEO -> System.out.println("It's the boss !");
                case CSuiteEmployee.CFO -> System.out.println("There's the money !");
                case RegularEmployee re -> System.out.println("A regular employee.");
                default -> System.out.println("An irregular employee.");
            }
        }
        default -> System.out.println("Some other kind of Id "
            + empObject.toString());
    }
}

```

Something may have occurred to you as you look at this code: since the **switch** variable can be any reference type, you can write many **case** statements that might be true for a given value of **switch** variable. In our example, we have a **switch** statement testing the kind of **Employee** object passed in. If you had two **case** statements, one pattern matching on the general **Employee** type

ncase Employee e -> ...

and one on the **RegularEmployee** type

ncase RegularEmployee re -> ...

and you passed in an object of type **RegularEmployee**, you might notice that both **case** statements would match. But, if so, in the context of your **switch** statement, which **case** would “win”? Only one **case** statement can be executed! To understand how which one is determined, you need to understand the concept of *case label dominance*.

One **case** label is said to *dominate* a second if it includes all the possible situations where the second applies and more. So, in our example, the **case Employee e** label dominates the **case RegularEmployee re** case label, because all **RegularEmployee** objects are also **Employee** objects, since the **RegularEmployee** class implements the **Employee** interface.

Now, Java uses a very simple approach to determine which one of possibly many **case** labels to select for a given value: it simply chooses the *first case* label in the order they are written that matches the **switch** variable. You need to understand **case** label dominance because this approach can potentially lead to cases where code could never be reached when one **case** label that dominates another appears before it in the **switch** statement. In fact, this situation will be interpreted by Java as a compilation error.

In our example earlier, in processing employee IDs that we know to be integers represented by the variable **i**, if you reversed the order of the **case** statements,

```
switch (i) {  
    case Integer idInt -> System.out.println("An employee Id: " + idInt);  
    case Integer idInt // compilation error!  
        when idInt < 0 -> System.out.println("A negative employee Id: ");  
    case -1, 0 -> System.out.println("A reserved employee Id");  
}
```

you would see that the first case label **Integer idInt** dominates the rest of the **case** labels that follow. All values of the **Integer i** would match the first **case** label, and the other case labels would never be reached in any circumstance. The code will not compile.

So when you write this kind of pattern matching **switch** statement, it is important to remember to put the least dominant **case** labels first. Case labels for a given type that use the contextual keyword **when** to refine the matching values are dominated by those that don't. In our example, **case**

Integer idInt dominates **case Integer idInt when idInt < 0** because the first applies to all **Integer** values, including all the negative ones. **case** labels using a pattern for a given type dominate **case** labels using a constant expression for that type. In our example code, the **case -1, 0** label is dominated by the other two **case** labels. In general, provided you remember to put the most specific **case** labels first in order, you should end up with code that compiles without errors.

With this final collection of features that have been added to **switch** expressions and statements, you can see now just how powerful they have become, particularly for concisely handling objects that may come in many shapes and forms.

Text Blocks

Beginning with JDK 15, Java provides support for *text blocks*. A text block is a new kind of literal that is comprised of a sequence of characters that can occupy more than one line. A text block reduces the tedium programmers often face when creating long string literals because newline characters can be used in a text block without the need for the `\n` escape sequence. Furthermore, tab and double quote characters can also be entered directly, without using an escape sequence, and the indentation of a multiline string can be preserved. Although text blocks may, at first, seem to be a relatively small addition to Java, they may well become one of the most popular features.

Text Block Fundamentals

A text block is supported by a new delimiter, which is three double-quote characters: `"""`. A text block is created by enclosing a string within a set of these delimiters. Specifically, a text block begins immediately following the newline after the opening `"""`. Thus, the line containing the opening delimiter must end with a newline. The text block begins on the next line. A text block ends at the first character of the closing `"""`. Here is a simple example:

```
"""
Text blocks make
multiple lines easy.
"""
```

This example creates a string in which the line "Text blocks make" is separated from "multiple lines easy." by a newline. It is not necessary to use the `\n` escape sequence to obtain the newline. Thus, the text block automatically preserves the newlines in the text. Again, the text block begins after the newline *following* the opening delimiter and ends at the start of the closing delimiter. Therefore, the newline after the second line is also preserved.

It is important to emphasize that even though a text block uses the "''''" delimiter, it is of type **String**. Thus, the preceding text block could be assigned to a **String** variable, as shown here:

```
String str = """
Text blocks make
multiple lines easy.
""" ;
```

When **str** is output using this statement:

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

the following is displayed:

```
Text blocks make
multiple lines easy.
```

Notice something else about this example. Because the last line ends with a newline, that newline will also be in the resulting string. If you don't want a trailing newline, then put the closing delimiter at the end of the last line, like this:

```
String str = """
Text blocks make
multiple lines easy."""; // now, no newline at the end
```

Understanding Leading Whitespace

In the preceding example, the text in the block was placed flush left. However, this is not required. You can have leading whitespace in a text block. There are two primary reasons that you might want leading whitespace. First, it will enable the text to be better aligned with the indentation level of the code around it. Second, it supports one or more levels of indentation within the text block itself.

In general, leading whitespace in a text block is automatically removed. However, the number of leading whitespaces to remove from each line is determined by the number of leading whitespaces in the line with the least indentation. For example, if all lines are flush left, then no whitespace is removed. If all lines are indented two spaces, then two spaces are removed from each line. However, if one line is indented two spaces, the next four spaces, and the third six spaces, then only two spaces are removed from the start of each line. This removes unwanted leading space while preserving the indentation of text within the block. This mechanism is illustrated by the following program:

```
// Demonstrate indentation in a text block.

class TextBlockDemo {

    public static void main(String[] args) {
        String str = """
                        Text blocks support strings that
                        span two or more lines and preserve
                        indentation. They reduce the
                        tedium associated with the
                        entry of long or complicated
                        strings into a program.
                    """;
        System.out.println(str);
    }
}
```

This program produces the following output:

```
Text blocks support strings that
span two or more lines and preserve
indentation. They reduce the
tedium associated with the
entry of long or complicated
strings into a program.
```

As you can see, leading whitespace has been removed up to, but not beyond, the level of the leftmost lines. Thus, the text block can be indented in the program to better fit the indentation level of the code, with the leading whitespace removed when the string is created. However, any whitespace after the indentation level of the block is preserved.

One other key point: The closing "''''" participates in determining the amount of whitespace to remove because it, too, can set the indentation level. Thus, if the closing delimiter is flush left, no whitespace is removed. Otherwise, whitespace is removed up to the first text character or when the closing delimiter is encountered. For example, consider this sequence:

```
String str = """
    A
        B
    C
"""; // this will determine the start of indent

String str2 = """
    A
        B
    C
"""; // this has no effect

String str3 = """
    A
        B
    C
"""; // this removes whitespace up to the """

System.out.print(str);
System.out.print(str2);
System.out.print(str3);
```

This sequence displays the following:

```
A  
B  
C  
A  
B  
C  
A  
B  
C
```

Pay special attention to the placement of the closing delimiter for **str2**. Because the number of preceding spaces for the lines containing A and C are fewer than that preceding the "'''", the closing delimiter has no effect on the number of spaces removed.

Use Double Quotes in a Text Block

Another major advantage to text blocks is the ability to use double quotes without the need for the \" escape sequence. In a text block, double quotes are treated like any other character. For example, consider the following program:

```
// Use double quotes in a text block.  
  
class TextBlockDemo2 {  
  
    public static void main(String[] args) {  
  
        String str = """  
                        A text block can use double quotes without  
                        the need for escape sequences. For example:  
  
                        He said, "The cat is on the roof."  
                        She asked, "How did it get up there?"  
                        """;  
  
        System.out.println(str);  
    }  
}
```

The output is shown here:

```
A text block can use double quotes without  
the need for escape sequences. For example:
```

```
He said, "The cat is on the roof."  
She asked, "How did it get up there?"
```

As you can see, there was no need to use the \" escape sequence. Furthermore, because double quotes are treated as “normal” characters, there is also no need for them to be balanced within a text block. For example

```
"""  
"xyz"  
"""
```

is perfectly acceptable. Just remember that three double quotes as a unit defines the text block delimiter.

Escape Sequences in Text Blocks

The escape sequences, such as \n or \t, can be used in a text block. However, because double quotes, newlines, and tabs can be entered directly, they will not often be needed. That said, with the addition of text blocks, two new escape sequences were added to Java. The first is \s, which specifies a space. Thus, it can be used to indicate trailing spaces. The second is \endofline. Because the \ must be followed by a line terminator, it must be used only at the end of the line. In a text block, the \ prevents a newline character from being included at the end of its line. Thus, the \ is essentially a line continuation indicator. For example:

```
String str = """  
    one \  
    two  
    three \  
    four  
    """;  
System.out.println(str);
```

Because the newline is suppressed after one and three, the output will be as shown here:

```
one two
three four
```

It is important to point out that `\endofline` can only be used in a text block. It cannot be used to continue a traditional string literal, for example.

One final point before leaving the topic of text blocks: Because text blocks provide a better, easier way to enter many types of strings into your program, it is anticipated that they will be widely used by the Java programmer community. It is likely that you will begin to encounter them in code that you work on, or use them in code that you create. Just remember, your JDK release must be at least 15 or later.

Pattern Matching with Records

JDK 21 introduces a convenient way to use pattern matching with records, called ***record patterns***. Given that a record defines the structure of its member variables, or ***components***, this feature simplifies the instanceof-then-cast idiom, tailored specifically to records. With record patterns, you can pass a record pattern as an operand of the **instanceof** operator, where a record pattern is of the general form

recordName(component-list)

and, most importantly, should the **instanceof** expression match, all the record components become available as local variables. Specifically, the record pattern form of **instanceof** is

objref instanceof recordName(component-list)

If the record pattern matches, each record component variable in the list will be automatically assigned the value of the corresponding record component from the matched *objref* object.

Let's work through an example to see firsthand the benefits record patterns can bring to your code. Let's return to processing objects

representing employees at a company using a Java record to encode their data.

```
record Employee(String name, Id id) {};
record Id(int idNum, Type type) {
    enum Type {FULL_TIME, PART_TIME}
}
```

This time you can see that there is some additional structure to represent different kinds of employees: full or part time. We can use record patterns to make it easy to process this kind of object in code. Now, assume you want to write some code to check if an object passed to you is an instance of this **Employee** record and, if it is, you will want to access some of its data, or record components. Prior to the advent of record patterns, you would use an **instanceof** expression with a type pattern and access the record data in the conditional block using its accessor methods. You can see this approach here:

```
if (obj instanceof Employee employee) {
    System.out.println("The employee ID for " +
        employee.name() + " is " +
        employee.id().idNum() + " and the type is " +
        employee.id().type()
    );
}
```

While it's clear what this code is doing, it also shows that the variable **employee** is simply being used as a holder to get at what you really are interested in from this record, which are its **name** and **id** components. With the record patterns feature, you can change the operand of **instanceof** to be a record pattern. In the event that the value is indeed a record of the type you specify, the record object is assigned to the variable you provide, and all its components become locally scoped variables that you can access. This holds true when the record components are simple types such as **String** or **Integer** and also for any class type or record whose nested structure matches the operand.

Here is a fuller listing using a record pattern to simplify the processing of the object passed to your code that may or may not be the **Employee**

object you want:

```
public static void printObject(Object obj) {  
    if (obj instanceof Employee(String empName, Id empId)) {  
        System.out.println("The employee ID for " +  
            empName + " is " +  
            empId.idNum() + " and the type is " +  
            empId.type());  
    } else if (obj == null) {  
        throw new IllegalArgumentException("null not allowed");  
    } else {  
        System.out.println("Printing: " + obj.toString());  
    }  
}
```

Notice that the component names in the record pattern, **empName** and **empId**, do not need to match names in the declaration of the record, **name** and **id**, in order for an **Employee** object to match the record pattern. This means you are free to name the components to suit the context of the code you are writing. An object that is **null** will never match a record pattern, even if its variable declaration is of the correct type. In this example, the **null** case is handled explicitly. You can see in the example that when the **obj** variable is indeed an **Employee** object, you don't have to use a variable for the **Employee** object; you can directly use its **name** and **id** components.

In fact, we can apply this pattern once more in our example because record patterns can nest inside each other. You'll notice in our example that while we got rid of the useless variable for the **Employee**, we still have the **empId** variable for the **Id** record, which is used only to get at the **idNum** and **type** values it holds. So, let's decompose this further with another record pattern for the **Id** record as well:

```

public static void printObjectNested(Object obj) {
    if (obj instanceof Employee(String empName, Id(id, Id.Type type))) {
        // here we can use the empName variable from the Employee object and
        // the id and type variables from the Employee objects Id object directly !
        System.out.println("The employee ID for " +
            empName + " is " +
            id + " and the type is " +
            type);
    } else if (obj == null) {
        throw new IllegalArgumentException("null not allowed");
    } else {
        System.out.println("Printing: " + obj.toString());
    }
}

```

Just as records are simply carriers for the data they hold, record patterns are an attractive way to simplify code so that it gets straight to the point of a record: its data.

Sealed Classes and Interfaces

Beginning with JDK 17, it is possible to declare a class that can be inherited by only specific subclasses. Such a class is called *sealed*. Prior to the advent of sealed classes, inheritance was an “all or nothing” situation. A class could either be extended by any subclass or marked as **final**, which prevented its inheritance entirely. Sealed classes fall between these two extremes because they enable you to specify precisely what subclasses a superclass will allow. In similar fashion, it is also possible to declare a sealed interface in which you specify only those classes that implement the interface and/or those interfaces that extend the sealed interface. Together, sealed classes and interfaces give you significantly greater control over inheritance, which can be especially important when designing class libraries.

Sealed Classes

To declare a sealed class, precede the declaration with **sealed**. Then, after the class name, include a **permits** clause that specifies the allowed subclasses. Both **sealed** and **permits** are context-sensitive keywords that

have special meaning only in a class or interface declaration. Outside of a class or interface declaration, **sealed** and **permits** are unrestricted and have no special meaning. Here is a simple example of a sealed class:

```
public sealed class MySealedClass permits Alpha, Beta {  
    // ...  
}
```

Here, the sealed class is called **MySealedClass**. It allows only two subclasses: **Alpha** and **Beta**. If any other class attempts to inherit **MySealedClass**, a compile-time error will occur.

Here are **Alpha** and **Beta**, the two subclasses of **MySealedClass**:

```
public final class Alpha extends MySealedClass {  
    // ...  
}  
  
public final class Beta extends MySealedClass {  
    // ...  
}
```

Notice that each is specified as **final**. In general, a subclass of a sealed class must be declared as either **final**, **sealed**, or **non-sealed**. Let's look at each option. First, in this example, each subclass is declared **final**. This means that the only subclasses of **MySealedClass** are **Alpha** and **Beta**, and no subclasses of either of those can be created. Therefore, the inheritance chain ends with **Alpha** and **Beta**.

To indicate that a subclass is itself sealed, it must be declared **sealed** and its permitted subclasses must be specified. For example, this version of **Alpha** permits **Gamma**:

```
public sealed class Alpha extends MySealedClass permits Gamma {  
    // ...  
}
```

Of course, the class **Gamma** must then itself be declared either **sealed**, **final**, or **non-sealed**.

At first it might seem a bit surprising, but you can unseal a subclass of a sealed class by declaring it **non-sealed**. This context-sensitive keyword was added by JDK 17. It unlocks the subclass, enabling it to be inherited by any other class. For example, **Beta** could be coded like this:

```
public non-sealed class Beta extends MySealedClass {  
    // ...  
}
```

Now, any class may inherit **Beta**. However, the only direct subclasses of **MySealedClass** remain **Alpha** and **Beta**. A primary reason for **non-sealed** is to enable a superclass to specify a limited set of direct subclasses that provide a baseline of well-defined functionality but allow those subclasses to be freely extended.

If a class is specified in a **permits** clause for a sealed class, then that class *must* directly extend the sealed class. Otherwise, a compile-time error will result. Thus, a sealed class and its subclasses define a mutually dependent logical unit. Additionally, it is illegal to declare a class that does not extend a sealed class as **non-sealed**.

A key requirement of a sealed class is that every subclass that it permits must be accessible. Furthermore, if a sealed class is contained in a named module, then each subclass must also be in the same named module. In this case, a subclass can be in a different package from the sealed class. If the sealed class is in the unnamed module, then the sealed class and all permitted subclasses must be in the same package.

In the preceding discussion, the superclass **MySealedClass** and its subclasses **Alpha** and **Beta** would have been stored in separate files because they are all public classes. However, it is also possible for a sealed class and its subclasses to be stored in a single file (formally, a compilation unit) as long as the subclasses have default package access. In cases such as this, no **permits** clause is required for a sealed class. For example, here all three classes are in the same file:

```
// Because this is all in one file, MySealedClass does not require
// a permits clause.
public sealed class MySealedClass {
    // ...
}

final class Alpha extends MySealedClass {
    // ...
}

final class Beta extends MySealedClass {
    // ...
}
```

One last point: An abstract class can also be sealed. There is no restriction in this regard.

Sealed Interfaces

A sealed interface is declared in the same way as a sealed class, by the use of **sealed**. A sealed interface uses its **permits** clause to specify the classes allowed to implement it and/or the interfaces allowed to extend it. Thus, a class that is not part of the **permits** clause cannot implement a sealed interface, and an interface not included in the **permits** clause cannot extend it.

Here is a simple example of a sealed interface that permits only the classes **Alpha** and **Beta** to implement it:

```
public sealed interface MySealedIF permits Alpha, Beta {
    void myMeth();
}
```

A class that implements a sealed interface must itself be specified as either **final**, **sealed**, or **non-sealed**. For example, here **Alpha** is marked **non-sealed** and **Beta** is specified as **final**:

```
public non-sealed class Alpha implements MySealedIF {
    public void myMeth() { System.out.println("In Alpha's myMeth()."); }
    // ...
}

public final class Beta implements MySealedIF {
    public void myMeth() { System.out.println("Inside Beta's myMeth()."); }
    // ...
}
```

Here is a key point: Any class specified in a sealed interface's **permits** clause *must* implement the interface. Thus, a sealed interface and its implementing classes form a logical unit.

A sealed interface can also specify which other interfaces can extend the sealed interface. For example, here, **MySealedIF** specifies that **MyIF** is permitted to extend it:

```
// Notice that MyIF is added to the permits clause.
public sealed interface MySealedIF permits Alpha, Beta, MyIF {
    void myMeth();
}
```

Because **MyIF** is part of the **MySealedIF permits** clause, it must be marked as either **non-sealed** or **sealed** and it must extend **MySealedIF**. For example:

```
public non-sealed interface MyIF extends MySealedIF {
    // ...
}
```

As you might expect, it is possible for a class to inherit a sealed class *and* implement a sealed interface. For example, here **Alpha** inherits **MySealedClass** and implements **MySealedIF**:

```
public non-sealed class Alpha extends MySealedClass implements MySealedIF {
    public void myMeth() { System.out.println("In Alpha's myMeth()."); }
    // ...
}
```

In the preceding examples, each class and interface are declared **public**. Thus, each is in its own file. However, as is the case with sealed classes, it is also possible for a sealed interface and its implementing classes (and extending interfaces) to be stored in a single file as long as the classes and interfaces have default package access. In cases such as this, no **permits** clause is required for a sealed interface. For example, here **MySealedIF** does not include a **permits** clause because **Alpha** and **Beta** are declared in the same file in the unnamed module:

```
public sealed interface MySealedIF {  
    void myMeth();  
}  
  
non-sealed class Alpha extends MySealedClass implements MySealedIF {  
    public void myMeth() { System.out.println("In Alpha's myMeth()."); }  
    // ...  
}  
  
final class Beta extends MySealedClass implements MySealedIF {  
    public void myMeth() { System.out.println("In Beta's myMeth()."); }  
    // ...  
}
```

One final point: Sealed classes and interfaces are most applicable to developers of API libraries in which subclasses and subinterfaces must be strictly controlled.

Future Directions

As discussed in [Chapter 1](#), beginning with JDK 12, Java releases may, and often do, include *preview features*. A preview feature is a new, fully developed enhancement to Java. However, a preview feature is *not yet* formally part of Java. Instead, a feature is previewed to allow programmers time to experiment with the feature and, if desired, communicate their thoughts and opinions prior to the feature being made permanent. This process enables a new feature to be improved or optimized based on actual developer use. As a result, a preview feature is *subject to change*. It can even be withdrawn. This means that a preview feature should not be used

for code that you intend to publicly release. That said, it is expected that most preview features will ultimately become part of Java, possibly after a period of refinement. Preview features chart the course of Java's future direction.

For example, the new pattern matching in **switch** and record pattern language features were first introduced in preview form in JDK 17 and JDK 19, respectively. They evolved in subsequent releases based on developer feedback, until they have reached their finalized forms, as described earlier in this chapter, in JDK 21. JDK 21 previews additional enhancements to the Java language, such as String Templates (JEP 430) and Unnamed Patterns and Variables (JEP 443), giving an indication of what may turn into finalized language features in a future release of the JDK. Because these are preview features that are subject to change, they are not discussed further in this book.

Java releases may also include *incubator modules*, which preview a new API or tool that is undergoing development. Like a preview feature, an incubator feature is subject to change. Furthermore, an incubator feature can be removed in the future. Thus, there is no guarantee that an incubator module will formally become part of Java in the future. Incubator features give developers an opportunity to experiment with the API or tool, and possibly supply feedback. JDK 21 includes one incubator module, the Vector API (JEP 448).

It is important to emphasize that preview features and incubator modules can be introduced in any Java release. Therefore, you will want to watch for them in each new version of Java. They give you a chance to try a new enhancement before it potentially becomes a formal part of Java. Perhaps more importantly, preview features and incubator modules give you advance information on where Java's development is headed.

PART

II

The Java Library

CHAPTER 18

String Handling

CHAPTER 19

Exploring java.lang

CHAPTER 20

java.util Part 1: The Collections Framework

CHAPTER 21

java.util Part 2: More Utility Classes

CHAPTER 22

Input/Output: Exploring java.io

CHAPTER 23

Exploring NIO

CHAPTER 24

Networking

CHAPTER 25

Event Handling

CHAPTER 26

Introducing the AWT: Working with Windows, Graphics, and Text

CHAPTER 27

Using AWT Controls, Layout Managers, and Menus

CHAPTER 28

Images

CHAPTER 29

The Concurrency Utilities

CHAPTER 30

The Stream API

CHAPTER 31

Regular Expressions and Other Packages

CHAPTER

18

String Handling

A brief overview of Java's string handling was presented in [Chapter 7](#). In this chapter, it is described in detail. As is the case in most other programming languages, in Java a string is a sequence of characters. But, unlike some other languages that implement strings as character arrays, Java implements strings as objects of type **String**.

Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. Also, **String** objects can be constructed a number of ways, making it easy to obtain a string when needed.

Somewhat unexpectedly, when you create a **String** object, you are creating a string that cannot be changed. That is, once a **String** object has been created, you cannot change the characters that comprise that string. At first, this may seem to be a serious restriction. However, such is not the case. You can still perform all types of string operations. The difference is that each time you need an altered version of an existing string, a new **String** object is created that contains the modifications. The original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is desired, Java provides two options: **StringBuffer** and **StringBuilder**. Both hold strings that can be modified after they are created.

The **String**, **StringBuffer**, and **StringBuilder** classes are defined in **java.lang**. Thus, they are available to all programs automatically. All are

declared **final**, which means that none of these classes may be subclassed. This allows certain optimizations that increase performance to take place on common string operations. All three implement the **CharSequence** interface.

One last point: To say that the strings within objects of type **String** are unchangeable means that the contents of the **String** instance cannot be changed after it has been created. However, a variable declared as a **String** reference can be changed to point at some other **String** object at any time.

The String Constructors

The **String** class supports several constructors. To create an empty **String**, call the default constructor. For example,

```
String s = new String();
```

will create an instance of **String** with no characters in it.

Frequently, you will want to create strings that have initial values. The **String** class provides a variety of constructors to handle this. To create a **String** initialized by an array of characters, use the constructor shown here:

String(char[] chars)

Here is an example:

```
char[] chars = { 'a', 'b', 'c' };
String s = new String(chars);
```

This constructor initializes **s** with the string "abc".

You can specify a subrange of a character array as an initializer using the following constructor:

String(char[] chars, int startIndex, int numChars)

Here, *startIndex* specifies the index at which the subrange begins, and *numChars* specifies the number of characters to use. Here is an example:

```
char[] chars = { 'a', 'b', 'c', 'd', 'e', 'f' };  
String s = new String(chars, 2, 3);
```

This initializes **s** with the characters **cde**.

You can construct a **String** object that contains the same character sequence as another **String** object using this constructor:

String(String strObj)

Here, *strObj* is a **String** object. Consider this example:

```
// Construct one String from another.  
class MakeString {  
    public static void main(String[] args) {  
        char[] c = {'J', 'a', 'v', 'a'};  
        String s1 = new String(c);  
        String s2 = new String(s1);  
  
        System.out.println(s1);  
        System.out.println(s2);  
    }  
}
```

The output from this program is as follows:

```
Java  
Java
```

As you can see, **s1** and **s2** contain the same string.

Even though Java's **char** type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set. Because 8-bit ASCII strings are common, the **String** class provides constructors that initialize a string when given a **byte** array. Two forms are shown here:

String(byte[] chrs)
String(byte[] chrs, int startIndex, int numChars)

Here, *chrs* specifies the array of bytes. The second form allows you to specify a subrange. In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform. The following program illustrates these constructors:

```
// Construct string from subset of char array.
class SubStringCons {
    public static void main(String[] args) {
        byte[] ascii = {65, 66, 67, 68, 69, 70};

        String s1 = new String(ascii);
        System.out.println(s1);

        String s2 = new String(ascii, 2, 3);
        System.out.println(s2);
    }
}
```

This program generates the following output:

```
ABCDEF
CDE
```

Starting with JDK 18, the default platform encoding is UTF-8, where previous JDKs chose an encoding that was dependent on the locale and operating system. Since UTF-8 has now become so widely adopted, this is usually the one you will want. But if you want to specify the encoding yourself, extended versions of the byte-to-string constructors are also defined in which you can specify the character encoding that determines how bytes are converted to characters. However, you will often want to use the default encoding provided by the platform.

NOTE The contents of the array are copied whenever you create a **String** object from an array. If you modify the contents of the array after you have created the string, the **String** will be unchanged.

You can construct a **String** from a **StringBuffer** by using the constructor shown here:

```
String(StringBuffer strBufObj)
```

You can construct a **String** from a **StringBuilder** by using this constructor:

```
String(StringBuilder strBuildObj)
```

The following constructor supports the extended Unicode character set:

```
String(int[ ] codePoints, int startIndex, int numChars)
```

Here, *codePoints* is an array that contains Unicode code points. The resulting string is constructed from the range that begins at *startIndex* and runs for *numChars*.

There are also constructors that let you specify a **Charset**.

NOTE A discussion of Unicode code points and how they are handled by Java is found in [Chapter 19](#).

String Length

The length of a string is the number of characters that it contains. To obtain this value, call the **length()** method, shown here:

```
int length()
```

The following fragment prints "3", since there are three characters in the string **s**:

```
char[] chars = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

Special String Operations

Because strings are a common and important part of programming, Java has added special support for several string operations within the syntax of the language. These operations include the automatic creation of new **String** instances from string literals, concatenation of multiple **String** objects by use of the + operator, and the conversion of other data types to a string representation. There are explicit methods available to perform all of

these functions, but Java does them automatically as a convenience for the programmer and to add clarity.

String Literals

The earlier examples showed how to explicitly create a **String** instance from an array of characters by using the **new** operator. However, there is an easier way to do this using a string literal. For each string literal in your program, Java automatically constructs a **String** object. Thus, you can use a string literal to initialize a **String** object. For example, the following code fragment creates two equivalent strings:

```
char[] chars = { 'a', 'b', 'c' };
String s1 = new String(chars);

String s2 = "abc"; // use string literal
```

Because a **String** object is created for every string literal, you can use a string literal any place you can use a **String** object. For example, you can call methods directly on a quoted string as if it were an object reference, as the following statement shows. It calls the **length()** method on the string "abc". As expected, it prints "3".

```
System.out.println("abc".length());
```

String Concatenation

In general, Java does not allow operators to be applied to **String** objects. The one exception to this rule is the + operator, which concatenates two strings, producing a **String** object as the result. This allows you to chain together a series of + operations. For example, the following fragment concatenates three strings:

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

This displays the string "He is 9 years old."

One practical use of string concatenation is found when you are creating very long strings. Instead of letting long strings wrap around within your source code, you can break them into smaller pieces, using the + to concatenate them. Here is an example:

```
// Using concatenation to prevent long lines.
class ConCat {
    public static void main(String[] args) {
        String longStr = "This could have been " +
            "a very long line that would have " +
            "wrapped around. But string concatenation " +
            "prevents this.";

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

String Concatenation with Other Data Types

You can concatenate strings with other types of data. For example, consider this slightly different version of the earlier example:

```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);
```

In this case, **age** is an **int** rather than another **String**, but the output produced is the same as before. This is because the **int** value in **age** is automatically converted into its string representation within a **String** object. This string is then concatenated as before. The compiler will convert an operand to its string equivalent whenever the other operand of the + is an instance of **String**.

Be careful when you mix other types of operations with string concatenation expressions, however. You might get surprising results. Consider the following:

```
String s = "four: " + 2 + 2;
System.out.println(s);
```

This fragment displays

```
four: 22
```

rather than the

```
four: 4
```

that you probably expected. Here's why: Operator precedence causes the concatenation of "four" with the string equivalent of 2 to take place first. This result is then concatenated with the string equivalent of 2 a second time. To complete the integer addition first, you must use parentheses, like this:

```
String s = "four: " + (2 + 2);
```

Now **s** contains the string "four: 4".

String Conversion and **toString()**

One way to convert data into its string representation is by calling one of the overloaded versions of the string conversion method **valueOf()** defined by **String**. **valueOf()** is overloaded for all the primitive types and for type **Object**. For the primitive types, **valueOf()** returns a string that contains the human-readable equivalent of the value with which it is called. For objects, **valueOf()** calls the **toString()** method on the object. We will look more closely at **valueOf()** later in this chapter. Here, let's examine the **toString()** method, because it is the means by which you can determine the string representation for objects of classes that you create.

Every class implements **toString()** because it is defined by **Object**. However, the default implementation of **toString()** is seldom sufficient. For most important classes that you create, you will want to override **toString()** and provide your own string representations. Fortunately, this is easy to do. The **toString()** method has this general form:

```
String toString()
```

To implement **toString()**, simply return a **String** object that contains the human-readable string that appropriately describes an object of your class.

By overriding **toString()** for classes that you create, you allow them to be fully integrated into Java's programming environment. For example, they can be used in **print()** and **println()** statements and in concatenation expressions. The following program demonstrates this by overriding **toString()** for the **Box** class:

```
// Override toString() for Box class.
class Box {
    double width;
    double height;
    double depth;

    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    public String toString() {
        return "Dimensions are " + width + " by " +
               depth + " by " + height + ".";
    }
}

class toStringDemo {
    public static void main(String[] args) {
        Box b = new Box(10, 12, 14);
        String s = "Box b: " + b; // concatenate Box object

        System.out.println(b); // convert Box to string
        System.out.println(s);
    }
}
```

The output of this program is shown here:

```
Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0
```

As you can see, **Box**'s **toString()** method is automatically invoked when a **Box** object is used in a concatenation expression or in a call to **println()**.

Character Extraction

The **String** class provides a number of ways in which characters can be extracted from a **String** object. Several are examined here. Although the characters that comprise a string within a **String** object cannot be indexed as if they were a character array, many of the **String** methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

charAt()

To extract a single character from a **String**, you can refer directly to an individual character via the **charAt()** method. It has this general form:

```
char charAt(int where)
```

Here, *where* is the index of the character that you want to obtain. The value of *where* must be nonnegative and specify a location within the string. **charAt()** returns the character at the specified location. For example,

```
char ch;
ch = "abc".charAt(1);
```

assigns the value **b** to **ch**.

getChars()

If you need to extract more than one character at a time, you can use the **getChars()** method. It has this general form:

```
void getChars(int sourceStart, int sourceEnd, char[ ] target, int
targetStart)
```

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from *sourceStart* through *sourceEnd*-1. The array that will receive the characters is specified by *target*. The index within *target* at which the substring will be copied is passed in *targetStart*. Care must be taken to ensure that the *target* array is large enough to hold the number of characters in the specified substring.

The following program demonstrates **getChars()**:

```
class getCharsDemo {  
    public static void main(String[] args) {  
        String s = "This is a demo of the getChars method.";  
        int start = 10;  
        int end = 14;  
        char[] buf = new char[end - start];  
  
        s.getChars(start, end, buf, 0);  
        System.out.println(buf);  
    }  
}
```

Here is the output of this program:

```
demo
```

getBytes()

There is an alternative to **getChars()** that stores the characters in an array of bytes. This method is called **getBytes()**, and it uses the default character-to-byte conversions provided by the platform. Here is its simplest form:

```
byte[ ] getBytes()
```

Other forms of **getBytes()** are also available. **getBytes()** is most useful when you are exporting a **String** value into an environment that does not support 16-bit Unicode characters.

toCharArray()

If you want to convert all the characters in a **String** object into a character array, the easiest way is to call **toCharArray()**. It returns an array of characters for the entire string. It has this general form:

```
char[ ] toCharArray( )
```

This function is provided as a convenience, since it is possible to use **getChars()** to achieve the same result.

String Comparison

The **String** class includes a number of methods that compare strings or substrings within strings. Several are examined here.

equals() and equalsIgnoreCase()

To compare two strings for equality, use **equals()**. It has this general form:

```
boolean equals(Object str)
```

Here, *str* is the **String** object being compared with the invoking **String** object. It returns **true** if the strings contain the same characters in the same order, and **false** otherwise. The comparison is case-sensitive.

To perform a comparison that ignores case differences, call **equalsIgnoreCase()**. When it compares two strings, it considers **A-Z** to be the same as **a-z**. It has this general form:

```
boolean equalsIgnoreCase(String str)
```

Here, *str* is the **String** object being compared with the invoking **String** object. It, too, returns **true** if the strings contain the same characters in the same order, and **false** otherwise.

Here is an example that demonstrates **equals()** and **equalsIgnoreCase()**:

```
// Demonstrate equals() and equalsIgnoreCase().  
class equalsDemo {  
    public static void main(String[] args) {  
        String s1 = "Hello";  
        String s2 = "Hello";  
        String s3 = "Good-bye";  
        String s4 = "HELLO";  
        System.out.println(s1 + " equals " + s2 + " -> " +  
                           s1.equals(s2));  
        System.out.println(s1 + " equals " + s3 + " -> " +  
                           s1.equals(s3));  
        System.out.println(s1 + " equals " + s4 + " -> " +  
                           s1.equals(s4));  
        System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +  
                           s1.equalsIgnoreCase(s4));  
    }  
}
```

The output from the program is shown here:

```
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true
```

regionMatches()

The **regionMatches()** method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows you to ignore case in such comparisons. Here are the general forms for these two methods:

```
boolean regionMatches(int startIndex, String str2,  
                     int str2startIndex, int numChars)
```

```
boolean regionMatches(boolean ignoreCase,  
                     int startIndex, String str2,  
                     int str2StartIndex, int numChars)
```

For both versions, *startIndex* specifies the index at which the region begins within the invoking **String** object. The **String** being compared is specified by *str2*. The index at which the comparison will start within *str2* is specified by *str2startIndex*. The length of the substring being compared is passed in *numChars*. In the second version, if *ignoreCase* is **true**, the case of the characters is ignored. Otherwise, case is significant.

startsWith() and endsWith()

String defines two methods that are, more or less, specialized forms of **regionMatches()**. The **startsWith()** method determines whether a given **String** begins with a specified string. Conversely, **endsWith()** determines whether the **String** in question ends with a specified string. They have the following general forms:

```
boolean startsWith(String str)  
boolean endsWith(String str)
```

Here, *str* is the **String** being tested. If the string matches, **true** is returned. Otherwise, **false** is returned. For example,

```
"Foobar".endsWith("bar")
```

and

```
"Foobar".startsWith("Foo")
```

are both **true**.

A second form of **startsWith()**, shown here, lets you specify a starting point:

```
boolean startsWith(String str, int startIndex)
```

Here, *startIndex* specifies the index into the invoking string at which point the search will begin. For example,

```
"Foobar".startsWith("bar", 3)
```

returns **true**.

equals() Versus ==

It is important to understand that the **equals()** method and the **==** operator perform two different operations. As just explained, the **equals()** method compares the characters inside a **String** object. The **==** operator compares two object references to see whether they refer to the same instance. The following program shows how two different **String** objects can contain the same characters, but references to these objects will not compare as equal:

```
// equals() vs ==
class EqualsNotEqualTo {
    public static void main(String[] args) {

        String s1 = "Hello";
        String s2 = new String(s1);

        System.out.println(s1 + " equals " + s2 + " -> " +
                           s1.equals(s2));
        System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
    }
}
```

The variable **s1** refers to the **String** instance created by "**Hello**". The object referred to by **s2** is created with **s1** as an initializer. Thus, the contents of the two **String** objects are identical, but they are distinct objects. This means that **s1** and **s2** do not refer to the same objects and are, therefore, not **==**, as is shown here by the output of the preceding example:

```
Hello equals Hello -> true
Hello == Hello -> false
```

compareTo()

Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is *less than*, *equal to*, or *greater than* the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after

the other in dictionary order. The method **compareTo()** serves this purpose. It is specified by the **Comparable<T>** interface, which **String** implements. It has this general form:

```
int compareTo(String str)
```

Here, *str* is the **String** being compared with the invoking **String**. The result of the comparison is returned and is interpreted as shown here:

Value	Meaning
Less than zero	The invoking string is less than <i>str</i> .
Greater than zero	The invoking string is greater than <i>str</i> .
Zero	The two strings are equal.

Here is a sample program that sorts an array of strings. The program uses **compareTo()** to determine sort ordering for a bubble sort:

```
// A bubble sort for Strings.
class SortString {
    static String[] arr = {
        "Now", "is", "the", "time", "for", "all", "good", "men",
        "to", "come", "to", "the", "aid", "of", "their", "country"
    };
    public static void main(String[] args) {
        for(int j = 0; j < arr.length; j++) {
            for(int i = j + 1; i < arr.length; i++) {
                if(arr[i].compareTo(arr[j]) < 0) {
                    String t = arr[j];
                    arr[j] = arr[i];

                    arr[i] = t;
                }
            }
            System.out.println(arr[j]);
        }
    }
}
```

The output of this program is the list of words:

```
Now
aid
all
come
country
for
good
is
men
of
the
the
their
time
to
to
```

As you can see from the output of this example, **compareTo()** takes into account uppercase and lowercase letters. The word "Now" came out before all the others because it begins with an uppercase letter, which means it has a lower value in the ASCII character set.

If you want to ignore case differences when comparing two strings, use **compareToIgnoreCase()**, as shown here:

```
int compareToIgnoreCase(String str)
```

This method returns the same results as **compareTo()**, except that case differences are ignored. You might want to try substituting it into the previous program. After doing so, "Now" will no longer be first.

Searching Strings

The **String** class provides two methods that allow you to search a string for a specified character or substring:

- **indexOf()** Searches for the first occurrence of a character or substring

- **lastIndexOf()** Searches for the last occurrence of a character or substring

These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or -1 on failure.

To search for the first occurrence of a character, use

```
int indexOf(int ch)
```

To search for the last occurrence of a character, use

```
int lastIndexOf(int ch)
```

Here, *ch* is the character being sought.

To search for the first or last occurrence of a substring, use

```
int indexOf(String str)  
int lastIndexOf(String str)
```

Here, *str* specifies the substring.

You can specify a starting point for the search using these forms:

```
int indexOf(int ch, int startIndex)  
int lastIndexOf(int ch, int startIndex)
```

```
int indexOf(String str, int startIndex)  
int lastIndexOf(String str, int startIndex)
```

Here, *startIndex* specifies the index at which point the search begins. For **indexOf()**, the search runs from *startIndex* to the end of the string. For **lastIndexOf()**, the search runs from *startIndex* to zero.

Beginning in JDK21, you can also specify an index range to narrow the search for the first occurrence of a character or substring using these forms:

```
int indexOf(String str, int startIndex, int endIndex)  
int indexOf(int ch, int startIndex, int endIndex)
```

Here, *startIndex* and *endIndex* specify the indices at which the search starts and ends, respectively. The first form searches for the substring given by *str*, the second for the character *ch*.

The following example shows how to use the various index methods to search inside of a **String**:

```
// Demonstrate indexOf() and lastIndexOf().  
class indexOfDemo {  
    public static void main(String[] args) {  
        String s = "Now is the time for all good men " +  
                  "to come to the aid of their country.";  
  
        System.out.println(s);  
        System.out.println("indexOf(t) = " +  
                           s.indexOf('t'));  
        System.out.println("lastIndexOf(t) = " +  
                           s.lastIndexOf('t'));  
        System.out.println("indexOf(the) = " +  
                           s.indexOf("the"));  
        System.out.println("lastIndexOf(the) = " +  
                           s.lastIndexOf("the"));  
        System.out.println("indexOf(t, 10) = " +  
                           s.indexOf('t', 10));  
        System.out.println("lastIndexOf(t, 60) = " +  
                           s.lastIndexOf('t', 60));  
  
        System.out.println("indexOf(the, 10) = " +  
                           s.indexOf("the", 10));  
        System.out.println("lastIndexOf(the, 60) = " +  
                           s.lastIndexOf("the", 60));  
        System.out.println("indexOf(the, 3, 47) = " +  
                           s.indexOf("the", 3, 47));  
    }  
}
```

Here is the output of this program:

```
Now is the time for all good men to come to the aid of their country.  
indexOf(t) = 7  
lastIndexOf(t) = 65  
indexOf(the) = 7  
lastIndexOf(the) = 55  
indexOf(t, 10) = 11  
lastIndexOf(t, 60) = 55  
indexOf(the, 10) = 44  
lastIndexOf(the, 60) = 55  
indexOf(the, 3, 47) = 7
```

Modifying a String

Because **String** objects are immutable, whenever you want to modify a **String**, you must either copy it into a **StringBuffer** or **StringBuilder**, or use a **String** method that constructs a new copy of the string with your modifications complete. A sampling of these methods are described here.

substring()

You can extract a substring using **substring()**. It has two forms. The first is

```
String substring(int startIndex)
```

Here, *startIndex* specifies the index at which the substring will begin. This form returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string.

The second form of **substring()** allows you to specify both the beginning and ending index of the substring:

```
String substring(int startIndex, int endIndex)
```

Here, *startIndex* specifies the beginning index, and *endIndex* specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

The following program uses **substring()** to replace all instances of one substring with another within a string:

```

// Substring replacement.
class StringReplace {
    public static void main(String[] args) {
        String org = "This is a test. This is, too.";
        String search = "is";

        String sub = "was";
        String result = "";
        int i;

        do { // replace all matching substrings
            System.out.println(org);
            i = org.indexOf(search);
            if(i != -1) {
                result = org.substring(0, i);
                result = result + sub;
                result = result + org.substring(i + search.length());
                org = result;
            }
        } while(i != -1);
    }
}

```

The output from this program is shown here:

```

This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.

```

concat()

You can concatenate two strings using **concat()**, shown here:

`String concat(String str)`

This method creates a new object that contains the invoking string with the contents of *str* appended to the end. **concat()** performs the same function as **+**. For example,

```
String s1 = "one";
String s2 = s1.concat("two");
```

puts the string "onetwo" into **s2**. It generates the same result as the following sequence:

```
String s1 = "one";
String s2 = s1 + "two";
```

replace()

The **replace()** method has two forms. The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:

String replace(char *original*, char *replacement*)

Here, *original* specifies the character to be replaced by the character specified by *replacement*. The resulting string is returned. For example,

```
String s = "Hello".replace('l', 'w');
```

puts the string "Hewwo" into **s**.

The second form of **replace()** replaces one character sequence with another. It has this general form:

String replace(CharSequence *original*, CharSequence *replacement*)

trim() and strip()

The **trim()** method returns a copy of the invoking string from which any leading and trailing spaces have been removed. As it relates to this method, spaces consist of those characters with a value of 32 or less. The **trim()** method has this general form:

String trim()

Here is an example:

```
String s = " Hello World ".trim();
```

This puts the string "Hello World" into `s`.

The `trim()` method is quite useful when you process user commands. For example, the following program prompts the user for the name of a state and then displays that state's capital. It uses `trim()` to remove any leading or trailing spaces that may have inadvertently been entered by the user.

```
// Using trim() to process commands.
import java.io.*;

class UseTrim {
    public static void main(String[] args)
        throws IOException
    {
        // create a BufferedReader using System.in
        BufferedReader br = new BufferedReader(new
            InputStreamReader(System.in, System.console().charset()));
        String str;

        System.out.println("Enter 'stop' to quit.");
        System.out.println("Enter State: ");
        do {
            str = br.readLine();
            str = str.trim(); // remove whitespace

            if(str.equals("Illinois"))
                System.out.println("Capital is Springfield.");
            else if(str.equals("Missouri"))
                System.out.println("Capital is Jefferson City.");
            else if(str.equals("California"))
                System.out.println("Capital is Sacramento.");
            else if(str.equals("Washington"))
                System.out.println("Capital is Olympia.");
            // ...
        } while(!str.equals("stop"));
    }
}
```

Beginning with JDK 11, Java also provides the methods **strip()**, **stripLeading()**, and **stripTrailing()**. The **strip()** method removes all whitespace characters (as defined by Java) from the beginning and end of the invoking string and returns the result. Such whitespace characters include, among others, spaces, tabs, carriage returns, and line feeds.

The methods **stripLeading()** and **stripTrailing()** delete whitespace characters from the start and end, respectively, of the invoking string and return the result. JDK 15 added the methods **stripIndent()**, which removes extraneous whitespace while retaining meaningful indentation, and **translateEscapes()**, which replaces escape sequences with their character equivalents.

Data Conversion Using **valueOf()**

The **valueOf()** method converts data from its internal format into a human-readable form. It is a static method that is overloaded within **String** for all of Java's built-in types so that each type can be converted properly into a string. **valueOf()** is also overloaded for type **Object**, so an object of any class type you create can also be used as an argument. (Recall that **Object** is a superclass for all classes.) Here are a few of its forms:

```
static String valueOf(double num)
static String valueOf(long num)
static String valueOf(Object ob)
static String valueOf(char[ ] chars)
```

As discussed earlier, **valueOf()** can be called when a string representation of some other type of data is needed. You can call this method directly with any data type and get a reasonable **String** representation. All of the simple types are converted to their common **String** representation. Any object that you pass to **valueOf()** will return the result of a call to the object's **toString()** method. In fact, you could just call **toString()** directly and get the same result.

For most arrays, **valueOf()** returns a rather cryptic string, which indicates that it is an array of some type. For arrays of **char**, however, a **String** object is created that contains the characters in the **char** array. There

is a special version of **valueOf()** that allows you to specify a subset of a **char** array. It has this general form:

```
static String valueOf(char[ ] chars, int startIndex, int numChars)
```

Here, *chars* is the array that holds the characters, *startIndex* is the index into the array of characters at which the desired substring begins, and *numChars* specifies the length of the substring.

Changing the Case of Characters Within a String

The method **toLowerCase()** converts all the characters in a string from uppercase to lowercase. The **toUpperCase()** method converts all the characters in a string from lowercase to uppercase. Nonalphabetical characters, such as digits, are unaffected. Here are the simplest forms of these methods:

```
String toLowerCase()  
String toUpperCase()
```

Both methods return a **String** object that contains the uppercase or lowercase equivalent of the invoking **String**. The default locale governs the conversion in both cases.

Here is an example that uses **toLowerCase()** and **toUpperCase()**:

```
// Demonstrate toUpperCase() and toLowerCase().  
  
class ChangeCase {  
    public static void main(String[] args)  
    {  
        String s = "This is a test."  
  
        System.out.println("Original: " + s);  
  
        String upper = s.toUpperCase();  
        String lower = s.toLowerCase();  
  
        System.out.println("Uppercase: " + upper);  
        System.out.println("Lowercase: " + lower);  
    }  
}
```

The output produced by the program is shown here:

```
Original: This is a test.  
Uppercase: THIS IS A TEST.  
Lowercase: this is a test.
```

One other point: Overloaded versions of **toLowerCase()** and **toUpperCase()** that let you specify a **Locale** object to govern the conversion are also supplied. Specifying the locale can be quite important in some cases and can help internationalize your application.

Joining Strings

The **join()** method is used to concatenate two or more strings, separating each string with a delimiter, such as a space or a comma. It has two forms. Its first is shown here:

```
static String join(CharSequence delim, CharSequence ... strs)
```

Here, *delim* specifies the delimiter used to separate the character sequences specified by *strs*. Because **String** implements the **CharSequence** interface, *strs* can be a list of strings. (See [Chapter 19](#) for information on

CharSequence.) The following program demonstrates this version of **join()**:

```
// Demonstrate the join() method defined by String.  
class StringJoinDemo {  
    public static void main(String[] args) {  
  
        String result = String.join(" ", "Alpha", "Beta", "Gamma");  
        System.out.println(result);  
  
        result = String.join(", ", "John", "ID#: 569",  
                             "E-mail: John@HerbSchildt.com");  
        System.out.println(result);  
    }  
}
```

The output is shown here:

```
Alpha Beta Gamma  
John, ID#: 569, E-mail: John@HerbSchildt.com
```

In the first call to **join()**, a space is inserted between each string. In the second call, the delimiter is a comma followed by a space. This illustrates that the delimiter need not be just a single character.

The second form of **join()** lets you join a list of strings obtained from an object that implements the **Iterable** interface. **Iterable** is implemented by the Collections Framework classes described in [Chapter 20](#), among others. See [Chapter 19](#) for information on **Iterable**.

Additional String Methods

In addition to those methods discussed earlier, **String** has many other methods. Several are summarized in the following table:

Method	Description
int codePointAt(int <i>i</i>)	Returns the Unicode code point at the location specified by <i>i</i> .
int codePointBefore(int <i>i</i>)	Returns the Unicode code point at the location that precedes that specified by <i>i</i> .
int codePointCount(int <i>start</i> , int <i>end</i>)	Returns the number of code points in the portion of the invoking String that are between <i>start</i> and <i>end</i> -1.
boolean contains(CharSequence <i>str</i>)	Returns true if the invoking object contains the string specified by <i>str</i> . Returns false otherwise.
boolean contentEquals(CharSequence <i>str</i>)	Returns true if the invoking string contains the same string as <i>str</i> . Otherwise, returns false .
boolean contentEquals(StringBuffer <i>str</i>)	Returns true if the invoking string contains the same string as <i>str</i> . Otherwise, returns false .
static String format(String <i>fmtstr</i> , Object ... <i>args</i>)	Returns a string formatted as specified by <i>fmtstr</i> . (See Chapter 20 for details on formatting.)
static String format(Locale <i>loc</i> , String <i>fmtstr</i> , Object ... <i>args</i>)	Returns a string formatted as specified by <i>fmtstr</i> . Formatting is governed by the locale specified by <i>loc</i> . (See Chapter 20 for details on formatting.)
String formatted(Object <i>args</i>)	Returns a string formatted as specified by the invoking string applied to <i>args</i> . (See Chapter 21 for details on formatting.)
String indent(int <i>num</i>)	When <i>num</i> is positive, indents each line in the invoking string by <i>num</i> spaces. When <i>num</i> is negative, each line has <i>num</i> leading whitespace characters deleted, if possible. Otherwise, when <i>num</i> is negative, leading whitespace is deleted until the first non-whitespace character is encountered. In all cases, including when <i>num</i> is zero, each line will end with a newline character. Returns the resulting string.
boolean isEmpty()	Returns true if the invoking string contains no characters and has a length of zero.

Method	Description
Stream<String> lines()	Decomposes a string into individual lines based on carriage return and line feed characters, and returns a Stream containing the lines.
boolean matches(string <i>regExp</i>)	Returns true if the invoking string matches the regular expression passed in <i>regExp</i> . Otherwise, returns false.
int offsetByCodePoints(int <i>start</i> , int <i>num</i>)	Returns the index within the invoking string that is <i>num</i> code points beyond the starting index specified by <i>start</i> .
String replaceFirst(String <i>regExp</i> , String <i>newStr</i>)	Returns a string in which the first substring that matches the regular expression specified by <i>regExp</i> is replaced by <i>newStr</i> .
String replaceAll(String <i>regExp</i> , String <i>newStr</i>)	Returns a string in which all substrings that match the regular expression specified by <i>regExp</i> are replaced by <i>newStr</i> .
String[] split(String <i>regExp</i>)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in <i>regExp</i> .
String[] split(String <i>regExp</i> , int <i>max</i>)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in <i>regExp</i> . The number of pieces is specified by <i>max</i> . If <i>max</i> is negative, then the invoking string is fully decomposed. Otherwise, if <i>max</i> contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If <i>max</i> is zero, the invoking string is fully decomposed, but no trailing empty strings will be included.
String[] splitWithDelimiters(String <i>regex</i> , int <i>limit</i>)	Decomposes the invoking string into parts and returns an array that contains the parts and the delimiters that match the regular expression passed in <i>regExp</i> . The number of parts returned is specified by <i>max</i> . If <i>max</i> is negative, then the invoking string is fully decomposed. Otherwise, if <i>max</i> contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If <i>max</i> is zero, the invoking string is fully decomposed.
CharSequence	Returns a substring of the invoking string, beginning

Notice that several of these methods work with regular expressions. Regular expressions are described in [Chapter 31](#). One other point, which is Beginning with JDK 12, **String** implements the **Constable** and **ConstantDesc** interfaces.

<code><R> R transform(Function<? super String, ? extends R> func)</code>	Executes the function specified by <i>func</i> on the invoking string and returns the result.
--	---

StringBuffer

StringBuffer supports a modifiable string. As you know, **String** represents fixed-length, immutable character sequences. In contrast, **StringBuffer** represents growable and writable character sequences. **StringBuffer** may have characters and substrings inserted in the middle or appended to the end. **StringBuffer** will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

StringBuffer Constructors

StringBuffer defines these four constructors:

```
StringBuffer()
StringBuffer(int size)
StringBuffer(String str)
StringBuffer(CharSequence chars)
```

The default constructor (the one with no parameters) reserves room for 16 characters without reallocation. The second version accepts an integer argument that explicitly sets the size of the buffer. The third version accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more characters without reallocation.

StringBuffer allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, **StringBuffer** reduces the number of reallocations that take place. The fourth constructor creates an object that contains the character sequence contained in *chars* and reserves room for 16 more characters.

length() and capacity()

The current length of a **StringBuffer** can be found via the **length()** method, while the total allocated capacity can be found through the **capacity()** method. They have the following general forms:

```
int length()
int capacity()
```

Here is an example:

```
// StringBuffer length vs. capacity.
class StringBufferDemo {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello");

        System.out.println("buffer = " + sb);
        System.out.println("length = " + sb.length());
        System.out.println("capacity = " + sb.capacity());
    }
}
```

Here is the output of this program, which shows how **StringBuffer** reserves extra space for additional manipulations:

```
buffer = Hello
length = 5
capacity = 21
```

Since **sb** is initialized with the string "Hello" when it is created, its length is 5. Its capacity is 21 because room for 16 additional characters is automatically added.

ensureCapacity()

If you want to preallocate room for a certain number of characters after a **StringBuffer** has been constructed, you can use **ensureCapacity()** to set the size of the buffer. This is useful if you know in advance that you will be

appending a large number of small strings to a **StringBuffer**.
ensureCapacity() has this general form:

```
void ensureCapacity(int minCapacity)
```

Here, *minCapacity* specifies the minimum size of the buffer. (A buffer larger than *minCapacity* may be allocated for reasons of efficiency.)

setLength()

To set the length of the string within a **StringBuffer** object, use **setLength()**. Its general form is shown here:

```
void setLength(int len)
```

Here, *len* specifies the length of the string. This value must be nonnegative.

When you increase the size of the string, null characters are added to the end. If you call **setLength()** with a value less than the current value returned by **length()**, then the characters stored beyond the new length will be lost. The **setCharAtDemo** sample program in the following section uses **setLength()** to shorten a **StringBuffer**.

charAt() and setCharAt()

The value of a single character can be obtained from a **StringBuffer** via the **charAt()** method. You can set the value of a character within a **StringBuffer** using **setCharAt()**. Their general forms are shown here:

```
char charAt(int where)
void setCharAt(int where, char ch)
```

For **charAt()**, *where* specifies the index of the character being obtained. For **setCharAt()**, *where* specifies the index of the character being set, and *ch* specifies the new value of that character. For both methods, *where* must be nonnegative and must not specify a location beyond the end of the string.

The following example demonstrates **charAt()** and **setCharAt()**:

```
// Demonstrate charAt() and setCharAt().  
class setCharAtDemo {  
  
    public static void main(String[] args) {  
        StringBuffer sb = new StringBuffer("Hello");  
        System.out.println("buffer before = " + sb);  
        System.out.println("charAt(1) before = " + sb.charAt(1));  
  
        sb.setCharAt(1, 'i');  
        sb.setLength(2);  
        System.out.println("buffer after = " + sb);  
        System.out.println("charAt(1) after = " + sb.charAt(1));  
    }  
}
```

Here is the output generated by this program:

```
public static void main(String[] args) {  
    StringBuffer sb = new StringBuffer("Hello");  
    System.out.println("buffer before = " + sb);  
    System.out.println("charAt(1) before = " + sb.charAt(1));  
  
    sb.setCharAt(1, 'i');  
    sb.setLength(2);  
    System.out.println("buffer after = " + sb);  
    System.out.println("charAt(1) after = " + sb.charAt(1));  
}
```

getChars()

To copy a substring of a **StringBuffer** into an array, use the **getChars()** method. It has this general form:

```
void getChars(int sourceStart, int sourceEnd, char[ ] target, int  
targetStart)
```

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. This means that the substring contains the characters from

sourceStart through *sourceEnd*-1. The array that will receive the characters is specified by *target*. The index within *target* at which the substring will be copied is passed in *targetStart*. Care must be taken to ensure that the *target* array is large enough to hold the number of characters in the specified substring.

append()

The **append()** method concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object. It has several overloaded versions. Here are a few of its forms:

```
buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i
```

First, the string representation of each parameter is obtained. Then, the result is appended to the current **StringBuffer** object. The buffer itself is returned by each version of **append()**. This allows subsequent calls to be chained together, as shown in the following example:

```
// Demonstrate append().
class appendDemo {
    public static void main(String[] args) {
        String s;
        int a = 42;
        StringBuffer sb = new StringBuffer(40);

        s = sb.append("a = ").append(a).append("!").toString();
        System.out.println(s);
    }
}
```

The output of this example is shown here:

```
a = 42!
```

insert()

The **insert()** method inserts one string into another. It is overloaded to accept values of all the primitive types, plus **Strings**, **Objects**, and **CharSequences**. Like **append()**, it obtains the string representation of the value it is called with. This string is then inserted into the invoking **StringBuffer** object. These are a few of its forms:

```
StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object obj)
```

Here, *index* specifies the index at which point the string will be inserted into the invoking **StringBuffer** object.

The following sample program inserts "like" between "I" and "Java":

```
// Demonstrate insert().
class insertDemo {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("I Java!");

        sb.insert(2, "like ");
        System.out.println(sb);
    }
}
```

The output of this example is shown here:

```
I like Java!
```

reverse()

You can reverse the characters within a **StringBuffer** object using **reverse()**, shown here:

```
StringBuffer reverse()
```

This method returns the reverse of the object on which it was called. The following program demonstrates **reverse()**:

```
// Using reverse() to reverse a StringBuffer.  
class ReverseDemo {  
    public static void main(String[] args) {  
        StringBuffer s = new StringBuffer("abcdef");  
  
        System.out.println(s);  
        s.reverse();  
  
        System.out.println(s);  
    }  
}
```

Here is the output produced by the program:

```
abcdef  
fedcba
```

delete() and deleteCharAt()

You can delete characters within a **StringBuffer** by using the methods **delete()** and **deleteCharAt()**. These methods are shown here:

```
StringBuffer delete(int startIndex, int endIndex)  
StringBuffer deleteCharAt(int loc)
```

The **delete()** method deletes a sequence of characters from the invoking object. Here, *startIndex* specifies the index of the first character to remove, and *endIndex* specifies an index one past the last character to remove. Thus, the substring deleted runs from *startIndex* to *endIndex*–1. The resulting **StringBuffer** object is returned.

The **deleteCharAt()** method deletes the character at the index specified by *loc*. It returns the resulting **StringBuffer** object.

Here is a program that demonstrates the **delete()** and **deleteCharAt()** methods:

```
// Demonstrate delete() and deleteCharAt()
class deleteDemo {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("This is a test.");
        sb.delete(4, 7);
        System.out.println("After delete: " + sb);

        sb.deleteCharAt(0);
        System.out.println("After deleteCharAt: " + sb);
    }
}
```

The following output is produced:

```
After delete: This a test.
After deleteCharAt: his a test.
```

replace()

You can replace one set of characters with another set inside a **StringBuffer** object by calling **replace()**. Its signature is shown here:

```
StringBuffer replace(int startIndex, int endIndex, String str)
```

The substring being replaced is specified by the indexes *startIndex* and *endIndex*. Thus, the substring at *startIndex* through *endIndex*–1 is replaced. The replacement string is passed in *str*. The resulting **StringBuffer** object is returned.

The following program demonstrates **replace()**:

```
// Demonstrate replace()
class replaceDemo {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("This is a test.");
        sb.replace(5, 7, "was");
        System.out.println("After replace: " + sb);
    }
}
```

Here is the output:

```
After replace: This was a test.
```

substring()

You can obtain a portion of a **StringBuffer** by calling **substring()**. It has the following two forms:

```
String substring(int startIndex)  
String substring(int startIndex, int endIndex)
```

The first form returns the substring that starts at *startIndex* and runs to the end of the invoking **StringBuffer** object. The second form returns the substring that starts at *startIndex* and runs through *endIndex*–1. These methods work just like those defined for **String** that were described earlier.

Additional StringBuffer Methods

In addition to those methods just described, **StringBuffer** supplies others. Several are summarized in the following table:

Method	Description
<code>StringBuffer appendCodePoint(int ch)</code>	Appends a Unicode code point to the end of the invoking object. A reference to the object is returned.
<code>int codePointAt(int i)</code>	Returns the Unicode code point at the location specified by <i>i</i> .
<code>int codePointBefore(int i)</code>	Returns the Unicode code point at the location that precedes that specified by <i>i</i> .
<code>int codePointCount(int start, int end)</code>	Returns the number of code points in the portion of the invoking String that are between <i>start</i> and <i>end</i> -1.
<code>int indexOf(String str)</code>	Searches the invoking StringBuffer for the first occurrence of <i>str</i> . Returns the index of the match, or -1 if no match is found.
<code>int indexOf(String str, int startIndex)</code>	Searches the invoking StringBuffer for the first occurrence of <i>str</i> , beginning at <i>startIndex</i> . Returns the index of the match, or -1 if no match is found.
<code>int lastIndexOf(String str)</code>	Searches the invoking StringBuffer for the last occurrence of <i>str</i> . Returns the index of the match, or -1 if no match is found.

Method	Description
<code>int lastIndexOf(String str, int startIndex)</code>	Searches the invoking StringBuffer for the last occurrence of <i>str</i> , beginning at <i>startIndex</i> . Returns the index of the match, or -1 if no match is found.
<code>int offsetByCodePoints(int start, int num)</code>	Returns the index within the invoking string that is <i>num</i> code points beyond the starting index specified by <i>start</i> .
<code>StringBuffer repeat(CharSequence cs, int n)</code>	Appends <i>n</i> copies of the supplied character sequence <i>cs</i> to the StringBuffer .
<code>StringBuffer repeat(int codePoint, int n)</code>	Appends <i>n</i> copies of the supplied Unicode code point to the StringBuffer .
<code>CharSequence subSequence(int startIndex, int stopIndex)</code>	Returns a substring of the invoking string, beginning at <i>startIndex</i> and stopping at <i>stopIndex</i> . This method is required by the CharSequence interface, which is implemented by StringBuffer .
<code>void trimToSize()</code>	Requests that the size of the character buffer for the invoking object be reduced to better fit the current contents.

The following program demonstrates **indexOf()** and **lastIndexOf()**:

```
class IndexOfDemo {  
    public static void main(String[] args) {  
        StringBuffer sb = new StringBuffer("one two one");  
        int i;  
  
        i = sb.indexOf("one");  
        System.out.println("First index: " + i);  
  
        i = sb.lastIndexOf("one");  
        System.out.println("Last index: " + i);  
    }  
}
```

The output is shown here:

```
First index: 0  
Last index: 8
```

StringBuilder

StringBuilder is similar to **StringBuffer**, except for one important difference: it is not synchronized, which means that it is not thread-safe. The advantage of **StringBuilder** is faster performance. However, in cases in which a mutable string will be accessed by multiple threads, and no external synchronization is employed, you must use **StringBuffer** rather than **StringBuilder**.

CHAPTER

19

Exploring `java.lang`

This chapter discusses classes and interfaces defined by **java.lang**. As you know, **java.lang** is automatically imported into all programs. It contains classes and interfaces that are fundamental to virtually all of Java programming. It is Java's most widely used package. Beginning with JDK 9, all of **java.lang** is part of the **java.base** module.

java.lang includes the following classes:

<code>Boolean</code>	<code>Float</code>	<code>ProcessBuilder</code>	<code>StringBuffer</code>
<code>Byte</code>	<code>InheritableThreadLocal</code>	<code>ProcessBuilder.Redirect</code>	<code>StringBuilder</code>
<code>Character</code>	<code>Integer</code>	<code>Record</code>	<code>System</code>
<code>Character.Subset</code>	<code>Long</code>	<code>Runtime</code>	<code>System.LoggerFinder</code>
<code>Character.UnicodeBlock</code>	<code>Math</code>	<code>RuntimePermission</code>	<code>Thread</code>
<code>Class</code>	<code>Module</code>	<code>Runtime.Version</code>	<code>ThreadGroup</code>
<code>ClassLoader</code>	<code>ModuleLayer</code>	<code>SecurityManager</code>	<code>ThreadLocal</code>
<code>ClassValue</code>	<code>ModuleLayer.Controller</code>	<code>Short</code>	<code>Throwable</code>
<code>Compiler</code>	<code>Number</code>	<code>StackTraceElement</code>	<code>Void</code>
<code>Double</code>	<code>Object</code>	<code>StackWalker</code>	
<code>Enum</code>	<code>Package</code>	<code>StrictMath</code>	
<code>Enum.EnumDesc</code>	<code>Process</code>	<code>String</code>	

java.lang defines the following interfaces:

Appendable	Iterable	StackWalker.StackFrame
AutoCloseable	ProcessHandle	System.Logger
CharSequence	ProcessHandle.Info	Thread.UncaughtExceptionHandler
Cloneable	Readable	
Comparable	Runnable	

Several of the classes contained in `java.lang` contain deprecated methods, many dating back to Java 1.0. Deprecated methods are still provided by Java to support legacy code but are not recommended for new code since they may be removed in a future release. Because of this, in most cases the deprecated methods are not discussed here.

Primitive Type Wrappers

As mentioned in Part I of this book, Java uses primitive types, such as `int` and `char`, for performance reasons. These data types are not part of the object hierarchy. They are passed by value to methods and cannot be directly passed by reference. Also, there is no way for two methods to refer to the *same instance* of an `int`. At times, you will need to create an object representation for one of these primitive types. For example, there are collection classes discussed in [Chapter 20](#) that deal only with objects; to store a primitive type in one of these classes, you need to wrap the primitive type in a class. To address this need, Java provides classes that correspond to each of the primitive types. In essence, these classes encapsulate, or *wrap*, the primitive types within a class. Thus, they are commonly referred to as *type wrappers*. The type wrappers were introduced in [Chapter 12](#). They are examined in detail here.

Before we begin, an important point needs to be mentioned. Beginning with JDK 16, the primitive type wrapper classes are now documented as *value-based*. As such, various rules and restrictions apply. For example, you should avoid using instances of a value-based class for synchronization. See [Chapter 13](#) for additional information on value-based classes.

Number

The abstract class **Number** defines a superclass that is implemented by the classes that wrap the numeric types **byte**, **short**, **int**, **long**, **float**, and **double**. **Number** has abstract methods that return the value of the object in each of the different number formats. For example, **doubleValue()** returns the value as a **double**, **floatValue()** returns the value as a **float**, and so on. These methods are shown here:

```
byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue()
```

The values returned by these methods might be rounded, truncated, or result in a “garbage” value due to the effects of a narrowing conversion.

Number has concrete subclasses that hold explicit values of each primitive numeric type: **Double**, **Float**, **Byte**, **Short**, **Integer**, and **Long**.

Double and Float

Double and **Float** are wrappers for floating-point values of type **double** and **float**, respectively. The constructors for **Float** are shown here:

```
Float(double num)
Float(float num)
Float(String str) throws NumberFormatException
```

Float objects can be constructed with values of type **float** or **double**. They can also be constructed from the string representation of a floating-point number. Beginning with JDK 9, these constructors have been deprecated, and beginning with JDK 16, they have been deprecated for removal. The **valueOf()** method is the strongly recommended alternative.

The constructors for **Double** are shown here:

```
Double(double num)
Double(String str) throws NumberFormatException
```

Double objects can be constructed with a **double** value or a string containing a floating-point value. Beginning with JDK 9, these constructors have been deprecated, and beginning with JDK 16, they have been deprecated for removal. The **valueOf()** method is the strongly recommended alternative.

The commonly used methods defined by **Float** include those shown in [Table 19-1](#). The commonly used methods defined by **Double** include those shown in [Table 19-2](#). Beginning with JDK 12, **Float** and **Double** also implement the **Constable** and **ConstantDesc** interfaces. Both **Float** and **Double** define the following constants:

Method	Description
byte byteValue()	Returns the value of the invoking object as a byte .
static int compare(float <i>num1</i> , float <i>num2</i>)	Compares the values of <i>num1</i> and <i>num2</i> . Returns 0 if the values are equal. Returns a negative value if <i>num1</i> is less than <i>num2</i> . Returns a positive value if <i>num1</i> is greater than <i>num2</i> .
int compareTo(Float <i>f</i>)	Compares the numerical value of the invoking object with that of <i>f</i> . Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
double doubleValue()	Returns the value of the invoking object as a double .
boolean equals(Object <i>FloatObj</i>)	Returns true if the invoking Float object is equivalent to <i>FloatObj</i> . Otherwise, it returns false .
static int floatToIntBits(float <i>num</i>)	Returns the IEEE-compatible, single-precision bit pattern that corresponds to <i>num</i> .
static int floatToRawIntBits(float <i>num</i>)	Returns the IEEE-compatible single-precision bit pattern that corresponds to <i>num</i> . A NaN value is preserved.
float floatValue()	Returns the value of the invoking object as a float .
int hashCode()	Returns the hash code for the invoking object.
static int hashCode(float <i>num</i>)	Returns the hash code for <i>num</i> .
static float intBitsToFloat(int <i>num</i>)	Returns float equivalent of the IEEE-compatible, single-precision bit pattern specified by <i>num</i> .
int intValue()	Returns the value of the invoking object as an int .
static boolean isFinite(float <i>num</i>)	Returns true if <i>num</i> is not NaN and is not infinite.
boolean isInfinite()	Returns true if the invoking object contains an infinite value. Otherwise, it returns false .
static boolean isInfinite(float <i>num</i>)	Returns true if <i>num</i> specifies an infinite value. Otherwise, it returns false .
boolean isNaN()	Returns true if the invoking object contains a value that is not a number. Otherwise, it returns false .

<code>static float sum(float val, float val2)</code>	Returns the result of <i>val</i> + <i>val2</i> .
<code>static String toHexString(float num)</code>	Returns a string containing the value of <i>num</i> in hexadecimal format.
<code>String toString()</code>	Returns the string equivalent of the invoking object.
<code>static String toString(float num)</code>	Returns the string equivalent of the value specified by <i>num</i> .
<code>static Float valueOf(float num)</code>	Returns a Float object containing the value passed in <i>num</i> .
<code>static Float valueOf(String str)</code> throws NumberFormatException	Returns the Float object that contains the value specified by the string in <i>str</i> .

Table 19-1 Commonly Used Methods Defined by **Float**

Method	Description
byte byteValue()	Returns the value of the invoking object as a byte .
static int compare(double <i>num1</i> , double <i>num2</i>)	Compares the values of <i>num1</i> and <i>num2</i> . Returns 0 if the values are equal. Returns a negative value if <i>num1</i> is less than <i>num2</i> . Returns a positive value if <i>num1</i> is greater than <i>num2</i> .
int compareTo(Double <i>d</i>)	Compares the numerical value of the invoking object with that of <i>d</i> . Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static long doubleToLongBits(double <i>num</i>)	Returns the IEEE-compatible, double-precision bit pattern that corresponds to <i>num</i> .
static long doubleToRawLongBits(double <i>num</i>)	Returns the IEEE-compatible double-precision bit pattern that corresponds to <i>num</i> . A NaN value is preserved.
double doubleValue()	Returns the value of the invoking object as a double .
boolean equals(Object <i>DoubleObj</i>)	Returns true if the invoking Double object is equivalent to <i>DoubleObj</i> . Otherwise, it returns false .
float floatValue()	Returns the value of the invoking object as a float .
int hashCode()	Returns the hash code for the invoking object.
static int hashCode(double <i>num</i>)	Returns the hash code for <i>num</i> .
int intValue()	Returns the value of the invoking object as an int .
static boolean isFinite(double <i>num</i>)	Returns true if <i>num</i> is not NaN and is not infinite.

<code>boolean isInfinite()</code>	Returns true if the invoking object contains an infinite value. Otherwise, it returns false .
<code>static boolean isInfinite(double num)</code>	Returns true if <i>num</i> specifies an infinite value. Otherwise, it returns false .
<code>boolean isNaN()</code>	Returns true if the invoking object contains a value that is not a number. Otherwise, it returns false .
<code>static boolean isNaN(double num)</code>	Returns true if <i>num</i> specifies a value that is not a number. Otherwise, it returns false .
<code>static double longBitsToDouble(long num)</code>	Returns double equivalent of the IEEE-compatible, double-precision bit pattern specified by <i>num</i> .
<code>long longValue()</code>	Returns the value of the invoking object as a long .
<code>static double max(double val, double val2)</code>	Returns the maximum of <i>val</i> and <i>val2</i> .
<code>static double min(double val, double val2)</code>	Returns the minimum of <i>val</i> and <i>val2</i> .
<code>static double parseDouble(String str) throws NumberFormatException</code>	Returns the double equivalent of the number contained in the string specified by <i>str</i> using radix 10.
<code>short shortValue()</code>	Returns the value of the invoking object as a short .
<code>static double sum(double val, double val2)</code>	Returns the result of <i>val</i> + <i>val2</i> .
<code>static String toHexString(double num)</code>	Returns a string containing the value of <i>num</i> in hexadecimal format.
<code>String toString()</code>	Returns the string equivalent of the invoking object.
<code>static String toString(double num)</code>	Returns the string equivalent of the value specified by <i>num</i> .
<code>static Double valueOf(double num)</code>	Returns a Double object containing the value passed in <i>num</i> .
<code>static Double valueOf(String str) throws NumberFormatException</code>	Returns a Double object that contains the value specified by the string in <i>str</i> .

Table 19-2 Commonly Used Methods Defined by **Double**

BYTES	The width of a float or double in bytes
MAX_EXPONENT	Maximum exponent
MAX_VALUE	Maximum positive value
MIN_EXPONENT	Minimum exponent
MIN_NORMAL	Minimum positive normal value
MIN_VALUE	Minimum positive value
NaN	Not a number
POSITIVE_INFINITY	Positive infinity
PRECISION	The number of bits used to represent this number type
NEGATIVE_INFINITY	Negative infinity
SIZE	The bit width of the wrapped value
TYPE	The Class object for float or double

The following example creates two **Double** objects—one by using a **double** value and the other by passing a string that can be parsed as a **double**:

```
class DoubleDemo {
    public static void main(String[] args) {
        Double d1 = Double.valueOf(3.14159);
        Double d2 = Double.valueOf("314159E-5");

        System.out.println(d1 + " = " + d2 + " -> " + d1.equals(d2));
    }
}
```

As you can see from the following output, both versions of **valueOf()** created identical **Double** instances, as shown by the **equals()** method returning **true**:

```
3.14159 = 3.14159 -> true
```

Understanding **isInfinite()** and **isNaN()**

Float and **Double** provide the methods **isInfinite()** and **isNaN()**, which help when manipulating two special **double** and **float** values. These methods test for two unique values: infinity and NaN (not a number). **isInfinite()** returns **true** if the value being tested is infinitely large or small in magnitude. **isNaN()** returns **true** if the value being tested is not a number.

The following example creates two **Double** objects; one is infinite, and the other is not a number:

```
// Demonstrate isInfinite() and isNaN()
class InfNaN {
    public static void main(String[] args) {
        Double d1 = Double.valueOf(1/0.);
        Double d2 = Double.valueOf(0/0.);

        System.out.println(d1 + ": " + d1.isInfinite() + ", " + d1isNaN());
        System.out.println(d2 + ": " + d2.isInfinite() + ", " + d2.isnan());
    }
}
```

This program generates the following output:

```
Infinity: true, false
NaN: false, true
```

Byte, Short, Integer, and Long

The **Byte**, **Short**, **Integer**, and **Long** classes are wrappers for **byte**, **short**, **int**, and **long** integer types, respectively. Their constructors are shown here:

`Byte(byte num)`

`Byte(String str)` throws `NumberFormatException`

`Short(short num)`

`Short(String str)` throws `NumberFormatException`

`Integer(int num)`

`Integer(String str)` throws `NumberFormatException`

`Long(long num)`

`Long(String str)` throws `NumberFormatException`

As you can see, these objects can be constructed from numeric values or from strings that contain valid whole number values. Beginning with JDK 9, these constructors have been deprecated, and beginning with JDK 16, they have been deprecated for removal. The `valueOf()` method is the strongly recommended alternative.

The commonly used methods defined by these classes are shown in [Tables 19-3](#) through [19-6](#). As you can see, they define methods for parsing integers from strings and converting strings back into integers. Variants of these methods allow you to specify the *radix*, or numeric base, for conversion. Common radices are 2 for binary, 8 for octal, 10 for decimal, and 16 for hexadecimal. Beginning with JDK 12, `Integer` and `Long` also implement the `Constable` and `ConstantDesc` interfaces. Beginning with JDK 15, `Byte` and `Short` also implement `Constable`.

Method	Description
byte byteValue()	Returns the value of the invoking object as a byte .
static int compare(byte <i>num1</i> , byte <i>num2</i>)	Compares the values of <i>num1</i> and <i>num2</i> . Returns 0 if the values are equal. Returns a negative value if <i>num1</i> is less than <i>num2</i> . Returns a positive value if <i>num1</i> is greater than <i>num2</i> .
int compareTo(Byte <i>b</i>)	Compares the numerical value of the invoking object with that of <i>b</i> . Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compareUnsigned(byte <i>num1</i> , byte <i>num2</i>)	Performs an unsigned comparison of <i>num1</i> and <i>num2</i> . Returns 0 if the values are equal. Returns a negative value if <i>num1</i> is less than <i>num2</i> . Returns a positive value if <i>num1</i> is greater than <i>num2</i> .
static Byte decode(String <i>str</i>) throws NumberFormatException	Returns a Byte object that contains the value specified by the string in <i>str</i> .
double doubleValue()	Returns the value of the invoking object as a double .
boolean equals(Object <i>ByteObj</i>)	Returns true if the invoking Byte object is equivalent to <i>ByteObj</i> . Otherwise, it returns false .
float floatValue()	Returns the value of the invoking object as a float .
int hashCode()	Returns the hash code for the invoking object.
static int hashCode(byte <i>num</i>)	Returns the hash code for <i>num</i> .
int intValue()	Returns the value of the invoking object as an int .
long longValue()	Returns the value of the invoking object as a long .
static byte parseByte(String <i>str</i>) throws NumberFormatException	Returns the byte equivalent of the number contained in the string specified by <i>str</i> using radix 10.
static byte parseByte(String <i>str</i> , int <i>radix</i>) throws NumberFormatException	Returns the byte equivalent of the number contained in the string specified by <i>str</i> using the specified radix.
short shortValue()	Returns the value of the invoking object as a short .
String toString()	Returns a string that contains the decimal equivalent of the invoking object.
static String toUnsignedString(byte <i>num</i>)	Returns a string that contains the decimal equivalent

Table 19-3 Commonly Used Methods Defined by `Byte`

	RETURNS a string that contains the decimal equivalent of <i>num</i> .
<code>static int toUnsignedInt(byte val)</code>	Returns the value of <i>val</i> as an unsigned integer.
<code>static long toUnsignedLong(byte val)</code>	Returns the value of <i>val</i> as an unsigned long integer.
<code>static Byte valueOf(byte num)</code>	Returns a <code>Byte</code> object containing the value passed in <i>num</i> .
<code>static Byte valueOf(String str)</code> <code>throws NumberFormatException</code>	Returns a <code>Byte</code> object that contains the value specified by the string in <i>str</i> .
<code>static Byte valueOf(String str, int radix)</code> <code>throws NumberFormatException</code>	Returns a <code>Byte</code> object that contains the value specified by the string in <i>str</i> using the specified <i>radix</i> .

Method	Description
byte byteValue()	Returns the value of the invoking object as a byte .
static int compare(short <i>num1</i> , short <i>num2</i>)	Compares the values of <i>num1</i> and <i>num2</i> . Returns 0 if the values are equal. Returns a negative value if <i>num1</i> is less than <i>num2</i> . Returns a positive value if <i>num1</i> is greater than <i>num2</i> .
int compareTo(Short <i>s</i>)	Compares the numerical value of the invoking object with that of <i>s</i> . Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compareUnsigned(short <i>num1</i> , short <i>num2</i>)	Performs an unsigned comparison of <i>num1</i> and <i>num2</i> . Returns 0 if the values are equal. Returns a negative value if <i>num1</i> is less than <i>num2</i> . Returns a positive value if <i>num1</i> is greater than <i>num2</i> .
static Short decode(String <i>str</i>) throws NumberFormatException	Returns a Short object that contains the value specified by the string in <i>str</i> .
double doubleValue()	Returns the value of the invoking object as a double .
boolean equals(Object <i>ShortObj</i>)	Returns true if the invoking Short object is equivalent to <i>ShortObj</i> . Otherwise, it returns false .
float floatValue()	Returns the value of the invoking object as a float .
int hashCode()	Returns the hash code for the invoking object.
static int hashCode(short <i>num</i>)	Returns the hash code for <i>num</i> .
int intValue()	Returns the value of the invoking object as an int .
long longValue()	Returns the value of the invoking object as a long .
static short parseShort(String <i>str</i>) throws NumberFormatException	Returns the short equivalent of the number contained in the string specified by <i>str</i> using radix 10.
static short parseShort(String <i>str</i> , int <i>radix</i>) throws NumberFormatException	Returns the short equivalent of the number contained in the string specified by <i>str</i> using the specified <i>radix</i> .
static short reverseBytes(short <i>num</i>)	Exchanges the high- and low-order bytes of <i>num</i> and returns the result.
short shortValue()	Returns the value of the invoking object as a short .
String toString()	Returns a string that contains the decimal equivalent of the invoking object.

<code>static String toString(short num)</code>	Returns a string that contains the decimal equivalent of <i>num</i> .
<code>static int toUnsignedInt(short val)</code>	Returns the value of <i>val</i> as an unsigned integer.
<code>static long toUnsignedLong(short val)</code>	Returns the value of <i>val</i> as an unsigned long integer.
<code>static Short valueOf(short num)</code>	Returns a Short object containing the value passed in <i>num</i> .
<code>static Short valueOf(String str) throws NumberFormatException</code>	Returns a Short object that contains the value specified by the string in <i>str</i> using radix 10.
<code>static Short valueOf(String str, int radix) throws NumberFormatException</code>	Returns a Short object that contains the value specified by the string in <i>str</i> using the specified <i>radix</i> .

Method	Description
static int bitCount(int <i>num</i>)	Returns the number of set bits in <i>num</i> .
byte byteValue()	Returns the value of the invoking object as a byte .
static int compare(int <i>num1</i> , int <i>num2</i>)	Compares the values of <i>num1</i> and <i>num2</i> . Returns 0 if the values are equal. Returns a negative value if <i>num1</i> is less than <i>num2</i> . Returns a positive value if <i>num1</i> is greater than <i>num2</i> .
int compareTo(Integer <i>i</i>)	Compares the numerical value of the invoking object with that of <i>i</i> . Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compareUnsigned(int <i>num1</i> , int <i>num2</i>)	Performs an unsigned comparison of <i>num1</i> and <i>num2</i> . Returns 0 if the values are equal. Returns a negative value if <i>num1</i> is less than <i>num2</i> . Returns a positive value if <i>num1</i> is greater than <i>num2</i> .
static Integer decode(String <i>str</i>) throws NumberFormatException	Returns an Integer object that contains the value specified by the string in <i>str</i> .
static int divideUnsigned(int <i>dividend</i> , int <i>divisor</i>)	Returns the result, as an unsigned value, of the unsigned division of <i>dividend</i> by <i>divisor</i> .
double doubleValue()	Returns the value of the invoking object as a double .
boolean equals(Object <i>IntegerObj</i>)	Returns true if the invoking Integer object is equivalent to <i>IntegerObj</i> . Otherwise, it returns false .
float floatValue()	Returns the value of the invoking object as a float .
static Integer getInteger(String <i>propertyName</i>)	Returns the value associated with the environmental property specified by <i>propertyName</i> . A null is returned on failure.
static Integer getInteger(String <i>propertyName</i> , int <i>default</i>)	Returns the value associated with the environmental property specified by <i>propertyName</i> . The value of <i>default</i> is returned on failure.
static Integer getInteger(String <i>propertyName</i> , Integer <i>default</i>)	Returns the value associated with the environmental property specified by <i>propertyName</i> . The value of <i>default</i> is returned on failure.
int hashCode()	Returns the hash code for the invoking object.
static int hashCode(int <i>num</i>)	Returns the hash code for <i>num</i> .
static int highestOneBit(int <i>num</i>)	Determines the position of the highest order set bit in <i>num</i> . It returns a value in which only this bit is set. If no bit is set to one, then zero is returned.

<code>static int numberOfTrailingZeros(int num)</code>	Returns the number of low-order zero bits that precede the first low-order set bit in <i>num</i> . If <i>num</i> is zero, 32 is returned.
<code>static int parseInt(CharSequence chars, int startIdx, int stopIdx, int radix) throws NumberFormatException</code>	Returns the integer equivalent of the number contained in the sequence specified by <i>chars</i> , between the indices <i>startIdx</i> and <i>stopIdx</i> -1, using the specified <i>radix</i> .
<code>static int parseInt(String str) throws NumberFormatException</code>	Returns the integer equivalent of the number contained in the string specified by <i>str</i> using radix 10.
<code>static int parseInt(String str, int radix) throws NumberFormatException</code>	Returns the integer equivalent of the number contained in the string specified by <i>str</i> using the specified <i>radix</i> .
<code>static int parseUnsignedInt(CharSequence chars, int startIdx, int stopIdx, int radix) throws NumberFormatException</code>	Returns the integer equivalent of the unsigned number contained in the sequence specified by <i>chars</i> , between the indices <i>startIdx</i> and <i>stopIdx</i> -1, using the specified <i>radix</i> .
<code>static int parseUnsignedInt(String str) throws NumberFormatException</code>	Returns the unsigned integer equivalent of the number contained in the string specified by <i>str</i> using the radix 10.
<code>static int parseUnsignedInt(String str, int radix) throws NumberFormatException</code>	Returns the unsigned integer equivalent of the number contained in the string specified by <i>str</i> using the radix specified by <i>radix</i> .
<code>static int remainderUnsigned(int dividend, int divisor)</code>	Returns the remainder, as an unsigned value, of the unsigned division of <i>dividend</i> by <i>divisor</i> .
<code>static int reverse(int num)</code>	Reverses the order of the bits in <i>num</i> and returns the result.
<code>static int reverseBytes(int num)</code>	Reverses the order of the bytes in <i>num</i> and returns the result.
<code>static int rotateLeft(int num, int n)</code>	Returns the result of rotating <i>num</i> left <i>n</i> positions.
<code>static int rotateRight(int num, int n)</code>	Returns the result of rotating <i>num</i> right <i>n</i> positions.
<code>short shortValue()</code>	Returns the value of the invoking object as a <code>short</code> .
<code>static int signum(int num)</code>	Returns -1 if <i>num</i> is negative, 0 if it is zero, and 1 if it is positive.
<code>static int sum(int val, int val2)</code>	Returns the result of <i>val</i> + <i>val2</i> .
<code>static String toBinaryString(int num)</code>	Returns a string that contains the binary equivalent of <i>num</i> .
<code>static String toHexString(int num)</code>	Returns a string that contains the hexadecimal equivalent of <i>num</i> .
<code>static String toOctalString(int num)</code>	Returns a string that contains the octal equivalent of <i>num</i> .

static String toUnsignedString(int <i>val</i> , int <i>radix</i>)	Returns a string that contains the value of <i>val</i> as an unsigned integer in the radix specified by <i>radix</i> .
static Integer valueOf(int <i>num</i>)	Returns an Integer object containing the value passed in <i>num</i> .
static Integer valueOf(String <i>str</i>) throws NumberFormatException	Returns an Integer object that contains the value specified by the string in <i>str</i> .
static Integer valueOf(String <i>str</i> , int <i>radix</i>) throws NumberFormatException	Returns an Integer object that contains the value specified by the string in <i>str</i> using the specified <i>radix</i> . <small>unsigned integer.</small>

Table 19-5 Commonly Used Methods Defined by **Integer**

Method	Description
static int bitCount(long <i>num</i>)	Returns the number of set bits in <i>num</i> .
byte byteValue()	Returns the value of the invoking object as a byte .
static int compare(long <i>num1</i> , long <i>num2</i>)	Compares the values of <i>num1</i> and <i>num2</i> . Returns 0 if the values are equal. Returns a negative value if <i>num1</i> is less than <i>num2</i> . Returns a positive value if <i>num1</i> is greater than <i>num2</i> .
int compareTo(Long <i>l</i>)	Compares the numerical value of the invoking object with that of <i>l</i> . Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compareUnsigned(long <i>num1</i> , long <i>num2</i>)	Performs an unsigned comparison of <i>num1</i> and <i>num2</i> . Returns 0 if the values are equal. Returns a negative value if <i>num1</i> is less than <i>num2</i> . Returns a positive value if <i>num1</i> is greater than <i>num2</i> .
static Long decode(String <i>str</i>) throws NumberFormatException	Returns a Long object that contains the value specified by the string in <i>str</i> .
static long divideUnsigned(long <i>dividend</i> , long <i>divisor</i>)	Returns the result, as an unsigned value, of the unsigned division of <i>dividend</i> by <i>divisor</i> .
double doubleValue()	Returns the value of the invoking object as a double .
boolean equals(Object <i>LongObj</i>)	Returns true if the invoking Long object is equivalent to <i>LongObj</i> . Otherwise, it returns false .
float floatValue()	Returns the value of the invoking object as a float .
static Long getLong(String <i>propertyName</i>)	Returns the value associated with the environmental property specified by <i>propertyName</i> . A null is returned on failure.
static Long getLong(String <i>propertyName</i> , long <i>default</i>)	Returns the value associated with the environmental property specified by <i>propertyName</i> . The value of <i>default</i> is returned on failure.
static Long getLong(String <i>propertyName</i> , Long <i>default</i>)	Returns the value associated with the environmental property specified by <i>propertyName</i> . The value of <i>default</i> is returned on failure.
int hashCode()	Returns the hash code for the invoking object.
static int hashCode(long <i>num</i>)	Returns the hash code for <i>num</i> .

<code>static long highestOneBit(long num)</code>	Determines the position of the highest-order set bit in <i>num</i> . It returns a value in which only this bit is set. If no bit is set to one, then zero is returned.
<code>int intValue()</code>	Returns the value of the invoking object as an <code>int</code> .
<code>long longValue()</code>	Returns the value of the invoking object as a <code>long</code> .
<code>static long lowestOneBit(long num)</code>	Determines the position of the lowest-order set bit in <i>num</i> . It returns a value in which only this bit is set. If no bit is set to one, then zero is returned.
<code>static long max(long val, long val2)</code>	Returns the maximum of <i>val</i> and <i>val2</i> .
<code>static long min(long val, long val2)</code>	Returns the minimum of <i>val</i> and <i>val2</i> .
<code>static int numberOfLeadingZeros(long num)</code>	Returns the number of high-order zero bits that precede the first high-order set bit in <i>num</i> . If <i>num</i> is zero, 64 is returned.
<code>static int numberOfTrailingZeros(long num)</code>	Returns the number of low-order zero bits that precede the first low-order set bit in <i>num</i> . If <i>num</i> is zero, 64 is returned.
<code>static long parseLong(CharSequence chars, int startIdx, int stopIdx, int radix)</code> <code>throws NumberFormatException</code>	Returns the <code>long</code> equivalent of the number contained in the sequence specified by <i>chars</i> , between the indices <i>startIdx</i> and <i>stopIdx</i> -1, using the specified <i>radix</i> .
<code>static long parseLong(String str) throws NumberFormatException</code>	Returns the <code>long</code> equivalent of the number contained in the string specified by <i>str</i> using radix 10.
<code>static long parseLong(String str, int radix) throws NumberFormatException</code>	Returns the <code>long</code> equivalent of the number contained in the string specified by <i>str</i> using the specified <i>radix</i> .
<code>static long parseUnsignedLong (CharSequence chars, int startIdx, int stopIdx, int radix)</code> <code>throws NumberFormatException</code>	Returns the <code>long</code> equivalent of the unsigned number contained in the sequence specified by <i>chars</i> , between the indices <i>startIdx</i> and <i>stopIdx</i> -1, using the specified <i>radix</i> .
<code>static long parseUnsignedLong(String str) throws NumberFormatException</code>	Returns the unsigned integer equivalent of the number contained in the string specified by <i>str</i> using the radix 10.
<code>static long parseUnsignedLong(String str, int radix) throws NumberFormatException</code>	Returns the unsigned integer equivalent of the number contained in the string specified by <i>str</i> using the radix specified by <i>radix</i> .
<code>static long remainderUnsigned (long dividend, long divisor)</code>	Returns the remainder, as an unsigned value, of the unsigned division of <i>dividend</i> by <i>divisor</i> .
<code>static long reverse(long num)</code>	Reverses the order of the bits in <i>num</i> and returns the result.

<code>static String toHexString(long num)</code>	Returns a string that contains the hexadecimal equivalent of <i>num</i> .
<code>static String toOctalString(long num)</code>	Returns a string that contains the octal equivalent of <i>num</i> .
<code>String toString()</code>	Returns a string that contains the decimal equivalent of the invoking object.
<code>static String toString(long num)</code>	Returns a string that contains the decimal equivalent of <i>num</i> .
<code>static String toString(long num, int radix)</code>	Returns a string that contains the decimal equivalent of <i>num</i> using the specified <i>radix</i> .
<code>static String toUnsignedString(long val)</code>	Returns a string that contains the decimal value of <i>val</i> as an unsigned integer.
<code>static String toUnsignedString(long val, int radix)</code>	Returns a string that contains the value of <i>val</i> as an unsigned integer in the radix specified by <i>radix</i> .
<code>static Long valueOf(long num)</code>	Returns a <code>Long</code> object containing the value passed in <i>num</i> .
<code>static Long valueOf(String str) throws NumberFormatException</code>	Returns a <code>Long</code> object that contains the value specified by the string in <i>str</i> .
<code>static Long valueOf(String str, int radix) throws NumberFormatException</code>	Returns a <code>Long</code> object that contains the value specified by the string in <i>str</i> using the specified <i>radix</i> .

Table 19-6 Commonly Used Methods Defined by `Long`

The following constants are defined:

<code>BYTES</code>	The width of the integer type in bytes
<code>MIN_VALUE</code>	Minimum value
<code>MAX_VALUE</code>	Maximum value
<code>SIZE</code>	The bit width of the wrapped value
<code>TYPE</code>	The <code>Class</code> object for <code>byte</code> , <code>short</code> , <code>int</code> , or <code>long</code>

Converting Numbers to and from Strings

One of the most common programming chores is converting the string representation of a number into its internal, binary format. Fortunately, Java provides an easy way to accomplish this. The `Byte`, `Short`, `Integer`, and `Long` classes provide the `parseByte()`, `parseShort()`, `parseInt()`, and `parseLong()` methods, respectively. These methods return the `byte`, `short`,

int, or **long** equivalent of the numeric string with which they are called.
(Similar methods also exist for the **Float** and **Double** classes.)

The following program demonstrates **parseInt()**. It sums a list of integers entered by the user. It reads the integers using **readLine()** and uses **parseInt()** to convert these strings into their **int** equivalents.

```
/* This program sums a list of numbers entered
   by the user.  It converts the string representation
   of each number into an int using parseInt().
*/
import java.io.*;
class ParseDemo {
    public static void main(String[] args)
        throws IOException
    {
        // create a BufferedReader using System.in
        BufferedReader br = new BufferedReader(new
            InputStreamReader(System.in, System.console().charset()));
        String str;
        int i;
        int sum=0;

        System.out.println("Enter numbers, 0 to quit.");
        do {
            str = br.readLine();
            try {
                i = Integer.parseInt(str);
            } catch(NumberFormatException e) {
                System.out.println("Invalid format");
                i = 0;
            }
            sum += i;
            System.out.println("Current sum is: " + sum);
        } while(i != 0);
    }
}
```

To convert a whole number into a decimal string, use the versions of **toString()** defined in the **Byte**, **Short**, **Integer**, or **Long** classes. The

Integer and **Long** classes also provide the methods **toBinaryString()**, **toHexString()**, and **toOctalString()**, which convert a value into a binary, hexadecimal, or octal string, respectively.

The following program demonstrates binary, hexadecimal, and octal conversion:

```
/* Convert an integer into binary, hexadecimal,
   and octal.

class StringConversions {
    public static void main(String[] args) {
        int num = 19648;
        System.out.println(num + " in binary: " +
                            Integer.toBinaryString(num));

        System.out.println(num + " in octal: " +
                            Integer.toOctalString(num));

        System.out.println(num + " in hexadecimal: " +
                            Integer.toHexString(num));
    }
}
```

The output of this program is shown here:

```
19648 in binary: 1001100110000000
19648 in octal: 46300
19648 in hexadecimal: 4cc0
```

Character

Character is a simple wrapper around a **char**. The constructor for **Character** is

Character(char *ch*)

Here, *ch* specifies the character that will be wrapped by the **Character** object being created. Beginning with JDK 9, this constructor has been

deprecated, and beginning with JDK 16, it has been deprecated for removal. The **valueOf()** method is the strongly recommended alternative.

To obtain the **char** value contained in a **Character** object, call **charValue()**, shown here:

```
char charValue()
```

It returns the character.

The **Character** class defines several constants, including the following:

BYTES	The width of a char in bytes
MAX_RADIX	The largest radix
MIN_RADIX	The smallest radix
MAX_VALUE	The largest character value
MIN_VALUE	The smallest character value
TYPE	The Class object for char

Character includes several static methods that categorize characters and alter their case. A sampling is shown in [Table 19-7](#). The following example demonstrates several of these methods:

Method	Description
static boolean isDefined(char <i>ch</i>)	Returns true if <i>ch</i> is defined by Unicode. Otherwise, it returns false .
static boolean isDigit(char <i>ch</i>)	Returns true if <i>ch</i> is a digit. Otherwise, it returns false .
static boolean isIdentifierIgnorable(char <i>ch</i>)	Returns true if <i>ch</i> should be ignored in an identifier. Otherwise, it returns false .
static boolean isISOControl(char <i>ch</i>)	Returns true if <i>ch</i> is an ISO control character. Otherwise, it returns false .
static boolean isJavaIdentifierPart(char <i>ch</i>)	Returns true if <i>ch</i> is allowed as part of a Java identifier (other than the first character). Otherwise, it returns false .
static boolean isJavaIdentifierStart(char <i>ch</i>)	Returns true if <i>ch</i> is allowed as the first character of a Java identifier. Otherwise, it returns false .
static boolean isLetter(char <i>ch</i>)	Returns true if <i>ch</i> is a letter. Otherwise, it returns false.
static boolean isLetterOrDigit(char <i>ch</i>)	Returns true if <i>ch</i> is a letter or a digit. Otherwise, it returns false .
static boolean isLowerCase(char <i>ch</i>)	Returns true if <i>ch</i> is a lowercase letter. Otherwise, it returns false .
static boolean isMirrored(char <i>ch</i>)	Returns true if <i>ch</i> is a mirrored Unicode character. A mirrored character is one that is reversed for text that is displayed right-to-left.
static boolean isSpaceChar(char <i>ch</i>)	Returns true if <i>ch</i> is a Unicode space character. Otherwise, it returns false .
static boolean isTitleCase(char <i>ch</i>)	Returns true if <i>ch</i> is a Unicode titlecase character. Otherwise, it returns false .
static boolean isUnicodeIdentifierPart(char <i>ch</i>)	Returns true if <i>ch</i> is allowed as part of a Unicode identifier (other than the first character). Otherwise, it returns false .
static Boolean isUnicodeIdentifierStart(char <i>ch</i>)	Returns true if <i>ch</i> is allowed as the first character of a Unicode identifier. Otherwise, it returns false .
static boolean isUpperCase(char <i>ch</i>)	Returns true if <i>ch</i> is an uppercase letter. Otherwise, it returns false .

Table 19-7 Various Character Methods

	Returns true if <i>ch</i> is whitespace. Otherwise, it returns false.
// Demonstrate several Is... methods.	
class IsDemo {	
public static void main(String[] args) {	
char[] a = {'a', 'b', '5', '?', 'A', ' '};	
for(int i=0; i<a.length; i++) {	
if(Character.isDigit(a[i]))	
System.out.println(a[i] + " is a digit.");	
if(Character.isLetter(a[i]))	
System.out.println(a[i] + " is a letter.");	
if(Character.isWhitespace(a[i]))	
System.out.println(a[i] + " is whitespace.");	
if(Character.isUpperCase(a[i]))	
System.out.println(a[i] + " is uppercase.");	
if(Character.isLowerCase(a[i]))	
System.out.println(a[i] + " is lowercase.");	
}	
}	
}	

The output from this program is shown here:

```
a is a letter.  
a is lowercase.  
b is a letter.  
b is lowercase.  
5 is a digit.  
A is a letter.  
A is uppercase.  
   is whitespace.
```

Character defines two methods, **forDigit()** and **digit()**, that enable you to convert between integer values and the digits they represent. They are shown here:

```
static char forDigit(int num, int radix)  
static int digit(char digit, int radix)
```

forDigit() returns the digit character associated with the value of *num*. The radix of the conversion is specified by *radix*. **digit()** returns the integer value associated with the specified character (which is presumably a digit) according to the specified radix. (There is a second form of **digit()** that takes a code point. See the following section for a discussion of code points.)

Another method defined by **Character** is **compareTo()**, which has the following form:

```
int compareTo(Character c)
```

It returns zero if the invoking object and *c* have the same value. It returns a negative value if the invoking object has a lower value. Otherwise, it returns a positive value.

Character includes a method called **getDirectionality()**, which can be used to determine the direction of a character. Several constants are defined that describe directionality. Most programs will not need to use character directionality.

Character also overrides **equals()** and **hashCode()**, and it provides a number of other methods. Beginning with JDK 15, **Character** also implements the **Constable** interface.

Two other character-related classes are **Character.Subset**, used to describe a subset of Unicode, and **Character.UnicodeBlock**, which contains Unicode character blocks.

Additions to Character for Unicode Code Point Support

A number of years ago, major additions were made to **Character** that support 32-bit Unicode characters. In the early days of Java, all Unicode characters could be held by 16 bits, which is the size of a **char** (and the size of the value encapsulated within a **Character**), because those values ranged from 0 to FFFF. However, the Unicode character set has been expanded, and more than 16 bits are required. Characters can now range from 0 to 10FFFF.

Here are three important terms. A *code point* is a character in the range 0 to 10FFFF. Characters that have values greater than FFFF are called *supplemental characters*. The *basic multilingual plane (BMP)* are those characters between 0 and FFFF.

The expansion of the Unicode character set caused a fundamental problem for Java. Because a supplemental character has a value greater than a **char** can hold, some means of handling the supplemental characters was needed. Java addressed this problem in two ways. First, Java uses two **chars** to represent a supplemental character. The first **char** is called the *high surrogate*, and the second is called the *low surrogate*. Methods, such as **codePointAt()**, were provided to translate between code points and supplemental characters.

Second, Java overloaded several preexisting methods in the **Character** class. The overloaded forms use **int** rather than **char** data. Because an **int** is large enough to hold any character as a single value, it can be used to store any character. For example, all of the methods in [Table 19-7](#) have overloaded forms that operate on **int**. Here is a sampling:

```
static boolean isDigit(int cp)
static boolean isLetter(int cp)
static int toLowerCase(int cp)
```

In addition to the methods overloaded to accept code points, **Character** adds methods that provide additional support for code points. A sampling is shown in [Table 19-8](#).

Method	Description
static int charCount(int <i>cp</i>)	Returns 1 if <i>cp</i> can be represented by a single char . It returns 2 if two chars are needed.
static int codePointAt(CharSequence <i>chars</i> , int <i>loc</i>)	Returns the code point at the location specified by <i>loc</i> .
static int codePointAt(char[] <i>chars</i> , int <i>loc</i>)	Returns the code point at the location specified by <i>loc</i> .
static int codePointBefore(CharSequence <i>chars</i> , int <i>loc</i>)	Returns the code point at the location that precedes that specified by <i>loc</i> .
static int codePointBefore(char[] <i>chars</i> , int <i>loc</i>)	Returns the code point at the location that precedes that specified by <i>loc</i> .
static boolean isBmpCodePoint(int <i>cp</i>)	Returns true if <i>cp</i> is part of the basic multilingual plane and false otherwise.
static boolean isHighSurrogate(char <i>ch</i>)	Returns true if <i>ch</i> contains a valid high surrogate character.
static boolean isLowSurrogate(char <i>ch</i>)	Returns true if <i>ch</i> contains a valid low surrogate character.
static boolean isSupplementaryCodePoint(int <i>cp</i>)	Returns true if <i>cp</i> contains a supplemental character.
static boolean isSurrogatePair(char <i>highCh</i> , char <i>lowCh</i>)	Returns true if <i>highCh</i> and <i>lowCh</i> form a valid surrogate pair.
static boolean isValidCodePoint(int <i>cp</i>)	Returns true if <i>cp</i> contains a valid code point.
static char[] toChars(int <i>cp</i>)	Converts the code point in <i>cp</i> into its char equivalent, which might require two chars . An array holding the result is returned.
static int toChars(int <i>cp</i> , char[] <i>target</i> , int <i>loc</i>)	Converts the code point in <i>cp</i> into its char equivalent, storing the result in <i>target</i> , beginning at <i>loc</i> . Returns 1 if <i>cp</i> can be represented by a single char . It returns 2 otherwise.
static int toCodePoint(char <i>highCh</i> , char <i>lowCh</i>)	Converts <i>highCh</i> and <i>lowCh</i> into their equivalent code point.

Table 19-8 A Sampling of Methods That Provide Support for 32-Bit Unicode Code Points

Boolean

Boolean is a very thin wrapper around **boolean** values, which is useful mostly when you want to pass a **boolean** variable by reference. It contains the constants **TRUE** and **FALSE**, which define true and false **Boolean** objects. **Boolean** also defines the **TYPE** field, which is the **Class** object for **boolean**. **Boolean** defines these constructors:

```
Boolean(boolean boolValue)  
Boolean(String boolString)
```

In the first version, *boolValue* must be either **true** or **false**. In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be **true**. Otherwise, it will be **false**. Beginning with JDK 9, these constructors have been deprecated, and beginning with JDK 16, they have been deprecated for removal. The **valueOf()** method is the strongly recommended alternative.

Commonly used methods defined by **Boolean** are shown in [Table 19-9](#). Beginning with JDK 15, **Boolean** also implements the **Constable** special-purpose interface.

Method	Description
boolean booleanValue()	Returns boolean equivalent.
static int compare(boolean <i>b1</i> , boolean <i>b2</i>)	Returns zero if <i>b1</i> and <i>b2</i> contain the same value. Returns a positive value if <i>b1</i> is true and <i>b2</i> is false . Otherwise, returns a negative value.
int compareTo(Boolean <i>b</i>)	Returns zero if the invoking object and <i>b</i> contain the same value. Returns a positive value if the invoking object is true and <i>b</i> is false . Otherwise, returns a negative value.
boolean equals(Object <i>boolObj</i>)	Returns true if the invoking object is equivalent to <i>boolObj</i> . Otherwise, it returns false .
static Boolean getBoolean(String <i>propertyName</i>)	Returns true if the system property specified by <i>propertyName</i> is true . Otherwise, it returns false .
int hashCode()	Returns the hash code for the invoking object.
static int hashCode(boolean <i>boolVal</i>)	Returns the hash code for <i>boolVal</i> .
static boolean logicalAnd(boolean <i>op1</i> , boolean <i>op2</i>)	Performs a logical AND of <i>op1</i> and <i>op2</i> and returns the result.
static boolean logicalOr(boolean <i>op1</i> , boolean <i>op2</i>)	Performs a logical OR of <i>op1</i> and <i>op2</i> and returns the result.
static boolean logicalXor(boolean <i>op1</i> , boolean <i>op2</i>)	Performs a logical XOR of <i>op1</i> and <i>op2</i> and returns the result.
static boolean parseBoolean(String <i>str</i>)	Returns true if <i>str</i> contains the string "true". Case is not significant. Otherwise, returns false .
String toString()	Returns the string equivalent of the invoking object.
static String toString(boolean <i>boolVal</i>)	Returns the string equivalent of <i>boolVal</i> .
static Boolean valueOf(boolean <i>boolVal</i>)	Returns the Boolean equivalent of <i>boolVal</i> .
static Boolean valueOf(String <i>boolString</i>)	Returns true if <i>boolString</i> contains the string "true" (in uppercase or lowercase). Otherwise, it returns false .

Table 19-9 Commonly Used Methods Defined by **Boolean**

Void

The **Void** class has one field, **TYPE**, which holds a reference to the **Class** object for type **void**. You do not create instances of this class.

Process

The abstract **Process** class encapsulates a *process*—that is, an executing program. It is used primarily as a superclass for the type of objects created by **exec()** in the **Runtime** class, or by **start()** in the **ProcessBuilder** class. A sampling of the **Process** methods are shown in [Table 19-10](#). Beginning with JDK 9, you can obtain a handle to the process in the form of a **ProcessHandle** instance, and you can obtain information about the process encapsulated in a **ProcessHandle.Info** instance. These offer additional control and information about a process. One particularly interesting piece of information is the amount of CPU time that a process receives. This is obtained by calling **totalCpuDuration()** defined by **ProcessHandle.Info**. Another especially helpful piece of information is obtained by calling **isAlive()** on a **ProcessHandle**. It will return **true** if the process is still executing. Beginning with JDK 17, **Process** also provides the methods **inputReader()**, **errorReader()**, and **outputWriter()**.

Method	Description
Stream<ProcessHandle> children()	Returns a stream that contains ProcessHandle objects that represent the immediate children of the invoking process.
Stream<ProcessHandle> descendants()	Returns a stream that contains ProcessHandle objects that represent both the immediate children of the invoking process, plus all of their descendants.
void destroy()	Terminates the process.
Process destroyForcibly()	Forces termination of the invoking process. Returns a reference to the process.
int exitValue()	Returns an exit code obtained from a subprocess.
InputStream getErrorStream()	Returns an input stream that reads input from the process's err output stream.
InputStream getInputStream()	Returns an input stream that reads input from the process's out output stream.
OutputStream getOutputStream()	Returns an output stream that writes output to the process's in input stream.
ProcessHandle.Info info()	Returns information about the process in the form of a ProcessHandle.Info object.
boolean isAlive()	Returns true if the invoking process is still active. Otherwise, returns false .
CompletableFuture<Process> onExit()	Returns a CompletableFuture for the invoking process, which can be used to perform tasks at termination.
long pid()	Returns the process ID associated with the invoking process.
boolean supportsNormalTermination()	Determines if a call to destroy() will result in normal or forced termination. Returns true if termination is normal, and false otherwise.
ProcessHandle toHandle()	Returns a handle to the invoking process in the form of a ProcessHandle object.
int waitFor() throws InterruptedException	Returns the exit code returned by the process. This method does not return until the process on which it is called terminates.
boolean waitFor(long waitTime, TimeUnit timeUnit) throws InterruptedException	Waits for the invoking process to end. The amount of time to wait is specified by waitTime in the units specified by timeUnit . Returns true if the process has ended and false if the wait time runs out.

Table 19-10 A Sampling of the Methods Defined by **Process**

Runtime

The **Runtime** class encapsulates the run-time environment. You cannot instantiate a **Runtime** object. However, you can get a reference to the current **Runtime** object by calling the static method **Runtime.getRuntime()**. Once you obtain a reference to the current **Runtime** object, you can call several methods that control the state and behavior of the Java Virtual Machine. Untrusted code typically cannot call any of the **Runtime** methods without raising a **SecurityException**. A sampling of methods defined by **Runtime** are shown in [Table 19-11](#).

Method	Description
<code>void addShutdownHook(Thread <i>thrd</i>)</code>	Registers <i>thrd</i> as a thread to be run when the Java Virtual Machine terminates.
<code>Process exec(String[] <i>comLineArray</i>) throws IOException</code>	Executes the command line specified by the strings in <i>comLineArray</i> as a separate process. An object of type <code>Process</code> is returned that describes the new process.
<code>Process exec(String[] <i>comLineArray</i>, String[] <i>environment</i>) throws IOException</code>	Executes the command line specified by the strings in <i>comLineArray</i> as a separate process with the environment specified by <i>environment</i> . An object of type <code>Process</code> is returned that describes the new process.
<code>void exit(int <i>exitCode</i>)</code>	Halts execution and returns the value of <i>exitCode</i> to the parent process. By convention, 0 indicates normal termination. All other values indicate some form of error.
<code>long freeMemory()</code>	Returns the approximate number of bytes of free memory available to the Java run-time system.
<code>void gc()</code>	Initiates garbage collection.
<code>static Runtime getRuntime()</code>	Returns the current <code>Runtime</code> object.
<code>void halt(int <i>code</i>)</code>	Immediately terminates the Java Virtual Machine. No shutdown hooks are run. The value of <i>code</i> is returned to the invoking process.
<code>void load(String <i>libraryFileName</i>)</code>	Loads the dynamic library whose file is specified by <i>libraryFileName</i> , which must specify its complete path.
<code>void loadLibrary(String <i>libraryName</i>)</code>	Loads the dynamic library whose name is associated with <i>libraryName</i> .
<code>Boolean removeShutdownHook(Thread <i>thrd</i>)</code>	Removes <i>thrd</i> from the list of threads to run when the Java Virtual Machine terminates. It returns <code>true</code> if successful—that is, if the thread was removed.
<code>long totalMemory()</code>	Returns the total number of bytes of memory available to the program.
<code>static Runtime.Version version()</code>	Returns the Java version being used. See <code>Runtime.Version</code> for details.

Table 19-11 A Sampling of Methods Defined by `Runtime`

NOTE JDK 18 deprecates the forms of `exec()` that take a `String` parameter representing a space-separated `String` of command arguments, preferring instead the equivalent forms that take a `String` array instead. This is due to ambiguities in how to parse parameters—for example those including filenames containing spaces.

Let's look at one of the more interesting uses of the `Runtime` class: executing additional processes.

Executing Other Programs

In safe environments, you can use Java to execute other heavyweight processes (that is, programs) on your multitasking operating system. Several forms of the `exec()` method allow you to name the program you want to run as well as its input parameters. The `exec()` method returns a `Process` object, which can then be used to control how your Java program interacts with this new running process. Because Java can run on a variety of platforms and under a variety of operating systems, `exec()` is inherently environment-dependent.

The following example uses `exec()` to launch `notepad`, Windows' simple text editor. Obviously, this example must be run under the Windows operating system.

There are several alternative forms of `exec()`, but the one shown in the example is often sufficient. The `Process` object returned by `exec()` can be manipulated by `Process`'s methods after the new program starts running. You can kill the subprocess with the `destroy()` method. The `waitFor()`

method causes your program to wait until the subprocess finishes. The `exitValue()` method returns the value returned by the subprocess when it is finished. This is typically 0 if no problems occur. Here is the preceding `exec()` example modified to wait for the running process to exit:

While a subprocess is running, you can write to and read from its standard input and output. The `getOutputStream()` and `getInputStream()` methods return the handles to standard `in` and `out` of the subprocess. Alternatively, beginning with JDK 17, you can also use `outputWriter()` and `inputReader()` to obtain a writer and reader. (I/O is examined in detail in [Chapter 22](#).)

Runtime.Version

Runtime.Version encapsulates version information (which includes the version number) pertaining to the Java environment. You can obtain an instance of **Runtime.Version** for the current platform by calling `Runtime.version()`. Originally added by JDK 9, **Runtime.Version** was substantially changed with the release of JDK 10 to better accommodate the faster, time-based release cadence. As discussed earlier in this book, starting with JDK 10, a feature release is anticipated to occur on a strict schedule, with the time between feature releases expected to be six months.

Runtime.Version is a value-based class. (See [Chapter 13](#) for a description of value-based classes.)

In the past, the JDK version number used the well-known *major.minor* approach. This mechanism did not, however, provide a good fit with the time-based release schedule. As a result, a different meaning was given to the elements of a version number. Today, the first four elements specify *counters*, which occur in the following order: feature release counter, interim release counter, update release counter, and patch release counter. Each number is separated by a period. However, trailing zeros, along with their preceding periods, are removed. Although additional elements may also be included, only the meaning of the first four are predefined.

The feature release counter specifies the number of the release. This counter is updated with each feature release. To smooth the transition from the previous version scheme, the feature release counter began at 10. Thus, the feature release counter for JDK 10 is 10, the one for JDK 11 is 11, and so on.

The interim release counter indicates the number of a release that occurs between feature releases. At the time of this writing, the value of the interim release counter will be zero because interim releases are not expected to be part of the increased release cadence. (It is defined for possible future use.) An interim release will not cause breaking changes to the JDK feature set. The update release counter indicates the number of a release that addresses security and possibly other problems. The patch release counter specifies a number of a release that addresses a serious flaw that must be fixed as soon as possible. With each new feature release, the interim, update, and patch counters are reset to zero.

It is useful to point out that the version number just described is a necessary component of the *version string*, but optional elements may also be included in the string. For example, a version string may include information for a pre-release version. Optional elements follow the version number in the version string.

Beginning with JDK 10, **Runtime.Version** was updated to include the following methods that support the new feature, interim, update, and patch counter values:

Each returns an integer value that represents the indicated value. Here is a short program that demonstrates their use:

As a result of the change to time-based releases, the following methods in **Runtime.Version** have been deprecated: **major()**, **minor()**, and **security()**. Previously, these returned the major version number, the minor version number, and the security update number. These values have been superseded by the feature, interim, and update numbers, as just described.

In addition to the methods just discussed, **Runtime.Version** has methods that obtain various pieces of optional data. For example, you can obtain the build number, if present, by calling **build()**. Pre-release information, if present, is returned by **pre()**. Other optional information may also be present and is obtained by calling **optional()**. You can compare versions by using **compareTo()** or **compareToOptional()**. You can use **equals()** and **equalsIgnoreOptional()** to determine version equality. The **version()** method returns a list of the version numbers. You can convert a valid version string into a **Runtime.Version** object by calling **parse()**.

ProcessBuilder

ProcessBuilder provides another way to start and manage processes (that is, programs). As explained earlier, all processes are represented by the **Process** class, and a process can be started by **Runtime.exec()**.

ProcessBuilder offers more control over the processes. For example, you can set the current working directory.

ProcessBuilder defines these constructors:

Here, *args* is a list of arguments that specify the name of the program to be executed along with any required command-line arguments. In the first constructor, the arguments are passed in a **List**. In the second, they are specified through a varargs parameter. [Table 19-12](#) describes the methods defined by **ProcessBuilder**.

Method	Description
List<String> command()	Returns a reference to a List that contains the name of the program and its arguments. Changes to this list affect the invoking object.
ProcessBuilder command(List<String> args)	Sets the name of the program and its arguments to those specified by <i>args</i> . Changes to this list affect the invoking object. Returns a reference to the invoking object.
ProcessBuilder command(String ... args)	Sets the name of the program and its arguments to those specified by <i>args</i> . Returns a reference to the invoking object.
File directory()	Returns the current working directory of the invoking object. This value will be null if the directory is the same as that of the Java program that started the process.
ProcessBuilder directory(File dir)	Sets the current working directory of the invoking object. Returns a reference to the invoking object.
Map<String, String> environment()	Returns the environmental variables associated with the invoking object as key/value pairs.
ProcessBuilder inheritIO()	Causes the invoked process to use the same source and target for the standard I/O streams as the invoking process.
ProcessBuilder.Redirect redirectError()	Returns the target for standard error as a ProcessBuilder.Redirect object.
ProcessBuilder redirectError(File f)	Sets the target for standard error to the specified file. Returns a reference to the invoking object.
ProcessBuilder redirectError(ProcessBuilder.Redirect target)	Sets the target for standard error as specified by <i>target</i> . Returns a reference to the invoking object.
boolean redirectErrorStream()	Returns true if the standard error stream has been redirected to the standard output stream. Returns false if the streams are separate.
ProcessBuilder redirectErrorStream(boolean merge)	If <i>merge</i> is true , then the standard error stream is redirected to standard output. If <i>merge</i> is false , the streams are separated, which is the default state. Returns a reference to the invoking object.

<code>ProcessBuilder redirectOutput(File f)</code>	Sets the target for standard output to the specified file. Returns a reference to the invoking object.
<code>ProcessBuilder redirectOutput(ProcessBuilder.Redirect target)</code>	Sets the target for standard output as specified by <i>target</i> . Returns a reference to the invoking object.
<code>Process start() throws IOException</code>	Begins the process specified by the invoking object. In other words, it runs the specified program.
<code>static List<Process> startPipeline(List<ProcessBuilder> pbList) throws IOException</code>	Pipelines the processes in <i>pbList</i> .

Table 19-12 The Methods Defined by **ProcessBuilder**

In [Table 19-12](#), notice the methods that use the **ProcessBuilder.Redirect** class. This abstract class encapsulates an I/O source or target linked to a subprocess. Among other things, these methods enable you to redirect the source or target of I/O operations. For example, you can redirect to a file by calling **to()**, redirect from a file by calling **from()**, and append to a file by calling **appendTo()**. A **File** object linked to the file can be obtained by calling **file()**. These methods are shown here:

Another method supported by **ProcessBuilder.Redirect** is **type()**, which returns a value of the enumeration type **ProcessBuilder.Redirect.Type**. This enumeration describes the type of the redirection. It defines these values: **APPEND**, **INHERIT**, **PIPE**, **READ**, and **WRITE**.

ProcessBuilder.Redirect also defines the constants **INHERIT**, **PIPE**, and **DISCARD**.

To create a process using **ProcessBuilder**, simply create an instance of **ProcessBuilder**, specifying the name of the program and any needed arguments. To begin execution of the program, call **start()** on that instance. Here is an example that executes the Windows text editor **notepad**. Notice that it specifies the name of the file to edit as an argument.

System

The **System** class holds a collection of static methods and variables. The standard input, output, and error output of the Java run time are stored in the **in**, **out**, and **err** variables, respectively. The non-deprecated methods defined by **System** are shown in [Table 19-13](#). Many of the methods throw a **SecurityException** if the operation is not permitted by the security manager. Be aware, however, that JDK 17 deprecates for removal the security manager.

Method	Description
static void arraycopy(Object <i>source</i> , int <i>sourceStart</i> , Object <i>target</i> , int <i>targetStart</i> , int <i>size</i>)	Copies an array. The array to be copied is passed in <i>source</i> , and the index at which point the copy will begin within <i>source</i> is passed in <i>sourceStart</i> . The array that will receive the copy is passed in <i>target</i> , and the index at which point the copy will begin within <i>target</i> is passed in <i>targetStart</i> . <i>size</i> is the number of elements that are copied.
static String clearProperty(String <i>which</i>)	Deletes the environmental variable specified by <i>which</i> . The previous value associated with <i>which</i> is returned.
static Console console()	Returns the console associated with the JVM. null is returned if the JVM currently has no console.
static long currentTimeMillis()	Returns the current time in terms of milliseconds since midnight, January 1, 1970.
static void exit(int <i>exitCode</i>)	Halts execution and returns the value of <i>exitCode</i> to the parent process (usually the operating system). By convention, 0 indicates normal termination. All other values indicate some form of error.
static void gc()	Initiates garbage collection.
static Map<String, String> getenv()	Returns a Map that contains the current environmental variables and their values.
static String getenv(String <i>which</i>)	Returns the value associated with the environmental variable passed in <i>which</i> .
static System.Logger getLogger(String <i>logName</i>)	Returns a reference to an object that can be used for program logging. The name of the logger is passed in <i>logName</i> .
static System.Logger getLogger(String <i>logName</i> , ResourceBundle <i>rb</i>)	Returns a reference to an object that can be used for program logging. The name of the logger is passed in <i>logName</i> . Localization is supported by the resource bundle passed in <i>rb</i> .
static Properties getProperties()	Returns the properties associated with the Java runtime system. (The Properties class is described in Chapter 20.)
static String getProperty(String <i>which</i>)	Returns the property associated with <i>which</i> . A null

<code>static void load(String <i>libraryFileName</i>)</code>	Loads the dynamic library whose file is specified by <i>libraryFileName</i> , which must specify its complete path.
<code>static void loadLibrary(String <i>libraryName</i>)</code>	Loads the dynamic library whose name is associated with <i>libraryName</i> .
<code>static String mapLibraryName(String <i>lib</i>)</code>	Returns a platform-specific name for the library named <i>lib</i> .
<code>static long nanoTime()</code>	Obtains the most precise timer in the system and returns its value in terms of nanoseconds since some arbitrary starting point. The accuracy of the timer is unknowable.
<code>static void setErr(PrintStream <i>eStream</i>)</code>	Sets the standard err stream to <i>eStream</i> .
<code>static void setIn(InputStream <i>iStream</i>)</code>	Sets the standard in stream to <i>iStream</i> .
<code>static void setOut(PrintStream <i>oStream</i>)</code>	Sets the standard out stream to <i>oStream</i> .
<code>static void setProperties(Properties <i>sysProperties</i>)</code>	Sets the current system properties as specified by <i>sysProperties</i> .
<code>static String setProperty(String <i>which</i>, String <i>v</i>)</code>	Assigns the value <i>v</i> to the property named <i>which</i> .

Table 19-13 The Non-Deprecated Methods Defined by **System**

Let's look at some common uses of **System**.

Using **currentTimeMillis()** to Time Program Execution

One use of the **System** class that you might find particularly interesting is to use the **currentTimeMillis()** method to time how long various parts of your program take to execute. The **currentTimeMillis()** method returns the current time in terms of milliseconds since midnight, January 1, 1970. To time a section of your program, store this value just before beginning the section in question. Immediately upon completion, call **currentTimeMillis()** again. The elapsed time will be the ending time minus the starting time. The following program demonstrates this:

Here is a sample run (remember that your results probably will differ):

If your system has a timer that offers nanosecond precision, then you could rewrite the preceding program to use **nanoTime()** rather than **currentTimeMillis()**. For example, here is the key portion of the program rewritten to use **nanoTime()**:

Using **arraycopy()**

The **arraycopy()** method can be used to copy quickly an array of any type from one place to another. This is much faster than the equivalent loop written out longhand in Java. Here is an example of two arrays being copied by the **arraycopy()** method. First, **a** is copied to **b**. Next, all of **a**'s elements are shifted *down* by one. Then, **b** is shifted *up* by one.

As you can see from the following output, you can copy using the same source and destination in either direction:

```
a = ABCDEFGHIJ
b = MMMMMMMMM
a = ABCDEFGHIJ
b = ABCDEFGHIJ
a = AABCDEFGH
b = BCDEFGHIJJ
```

Environment Properties

At the time of this writing, the following properties are available:

file.separator	java.vendor	java.vm.version
java.class.path	java.vendor.url	line.separator
java.class.version	java.vendor.version	native.encoding
java.compiler	java.version	os.arch
java.home	java.version.date	os.name
java.io.tmpdir	java.vm.name	os.version
java.library.path	java.vm.specification.name	path.separator
java.specification.name	java.vm.specification.vendor	user.dir
java.specification.vendor	java.vm.specification.version	user.home
java.specification.version	java.vm.vendor	user.name

You can obtain the values of various environment variables by calling the **System.getProperty()** method. For example, the following program displays the path to the current user directory:

System.Logger and System.LoggerFinder

The **System.Logger** interface and **System.LoggerFinder** class support a program log. A logger can be found by use of **System.getLogger()**. **System.Logger** provides the interface to the logger.

Object

As mentioned in Part I, **Object** is a superclass of all other classes. **Object** defines the methods shown in [Table 19-14](#), which are available to every object.

Method	Description
Object clone() throws CloneNotSupportedException	Creates a new object that is the same as the invoking object.
boolean equals(Object <i>object</i>)	Returns true if the invoking object is equivalent to <i>object</i> .
void finalize() throws Throwable	Default finalize() method. It is called before an unused object is recycled. (Deprecated for removal by JDK 18.)
final Class<?> getClass()	Obtains a Class object that describes the invoking object.
int hashCode()	Returns the hash code associated with the invoking object.
final void notify()	Notifies a thread waiting on the invoking object.
final void notifyAll()	Notifies all threads waiting on the invoking object.
String toString()	Returns a string that describes the object.
final void wait() throws InterruptedException	Waits on another thread of execution.
final void wait(long <i>milliseconds</i>) throws InterruptedException	Waits up to the specified number of <i>milliseconds</i> on another thread of execution.
final void wait(long <i>milliseconds</i> , int <i>nanoseconds</i>) throws InterruptedException	Waits up to the specified number of <i>milliseconds</i> plus <i>nanoseconds</i> on another thread of execution.

Table 19-14 The Methods Defined by **Object**

Using **clone()** and the **Cloneable** Interface

Most of the methods defined by **Object** are discussed elsewhere in this book. However, one deserves special attention: **clone()**. The **clone()** method generates a duplicate copy of the object on which it is called. Only classes that implement the **Cloneable** interface can be cloned.

The **Cloneable** interface defines no members. It is used to indicate that a class allows an exact copy of an object (that is, a *clone*) to be made. If you try to call **clone()** on a class that does not implement **Cloneable**, a **CloneNotSupportedException** is thrown. When a clone is made, the

constructor for the object being cloned is *not* called. As implemented by **Object**, a clone is simply an exact copy of the original.

Cloning is a potentially dangerous action, because it can cause unintended side effects. For example, if the object being cloned contains a reference variable called *obRef*, then when the clone is made, *obRef* in the clone will refer to the same object as does *obRef* in the original. If the clone makes a change to the contents of the object referred to by *obRef*, then it will be changed for the original object, too. Here is another example: If an object opens an I/O stream and is then cloned, two objects will be capable of operating on the same stream. Further, if one of these objects closes the stream, the other object might still attempt to write to it, causing an error. In some cases, you will need to override the **clone()** method defined by **Object** to handle these types of problems.

Because cloning can cause problems, **clone()** is declared as **protected** inside **Object**. This means that it must either be called from within a method defined by the class that implements **Cloneable**, or it must be explicitly overridden by that class so that it is public. Let's look at an example of each approach.

The following program implements **Cloneable** and defines the method **cloneTest()**, which calls **clone()** in **Object**:

Here, the method **cloneTest()** calls **clone()** in **Object** and returns the result. Notice that the object returned by **clone()** must be cast into its appropriate type (**TestClone**).

The following example overrides **clone()** so that it can be called from code outside of its class. To do this, its access specifier must be **public**, as shown here:

The side effects caused by cloning are sometimes difficult to see at first. It is easy to think that a class is safe for cloning when it actually is not. In general, you should not implement **Cloneable** for any class without good reason.

Class

Class encapsulates the run-time state of a class or interface. Objects of type **Class** are created automatically, when classes are loaded. You cannot explicitly declare a **Class** object. Generally, you obtain a **Class** object by calling the **getClass()** method defined by **Object**. **Class** is a generic class that is declared as shown here:

```
class Class<T>
```

Here, **T** is the type of the class or interface represented. A sampling of methods defined by **Class** is shown in [Table 19-15](#). In the table, notice the **getModule()** method. It is part of the support for the modules feature added by JDK 9. **Class** implements several interfaces, including **Constable** and **TypeDescriptor**.

Method	Description
static Class<?> forName(Module <i>mod</i> , String <i>name</i>)	Returns a Class object corresponding to its complete name and the module in which it resides.
static Class<?> forName(String <i>name</i>) throws ClassNotFoundException	Returns a Class object given its complete name.
static Class<?> forName(String <i>name</i> , boolean <i>how</i> , ClassLoader <i>ldr</i>) throws ClassNotFoundException	Returns a Class object given its complete name. The object is loaded using the loader specified by <i>ldr</i> . If <i>how</i> is true , the object is initialized; otherwise, it is not.
<A extends Annotation> A getAnnotation(Class<A> <i>annoType</i>)	Returns an Annotation object that contains the annotation associated with <i>annoType</i> for the invoking object.
Annotation[] getAnnotations()	Obtains all annotations associated with the invoking object and stores them in an array of Annotation objects. Returns a reference to this array.
<A extends Annotation> A[] getAnnotationsByType(Class<A> <i>annoType</i>)	Returns an array of the annotations (including repeated annotations) of <i>annoType</i> associated with the invoking object.

<code>Method[] getMethods() throws SecurityException</code>	Obtains a Method object for each public method of the class or interface represented by the invoking object and stores them in an array. Returns a reference to this array.
<code>Module getModule()</code>	Returns a Module object that represents the module in which the invoking class type resides.
<code>String getName()</code>	Returns the complete name of the class or interface of the type represented by the invoking object.
<code>String getPackageName()</code>	Returns the name of the package of which the invoking class type is a part.
<code>ProtectionDomain getProtectionDomain()</code>	Returns the protection domain associated with the invoking object.
<code>Class<? super T> getSuperclass()</code>	Returns the superclass of the type represented by the invoking object. The return value is <code>null</code> if the represented type is Object or not a class.
<code>boolean isInterface()</code>	Returns <code>true</code> if the type represented by the invoking object is an interface. Otherwise, it returns <code>false</code> .
<code>String toString()</code>	Returns the string representation of the type represented by the invoking object or interface.

Table 19-15 A Sampling of Methods Defined by **Class**

The methods defined by **Class** are often useful in situations where runtime type information about an object is required. As [Table 19-15](#) shows, methods are provided that allow you to determine additional information about a particular class, such as its public constructors, fields, and methods. Among other things, this is important for the Java Beans functionality, which is discussed later in this book.

The following program demonstrates `getClass()` (inherited from **Object**) and `getSuperclass()` (from **Class**):

The output from this program is shown here:

Beginning with JDK 16, **Class** has included methods that support records. These are **getRecordComponents()**, which obtains information about a record's components, and **isRecord()**, which returns true if the invoking **Class** represents a record. Beginning with JDK 17, **Class** has

included methods that support sealed classes and interfaces. They are `isSealed()`, which returns true if the invoking **Class** is a sealed, and `getPermittedSubclasses()`, which obtains an array of **Class** instances of the subclasses or subinterfaces permitted by the invoking **Class**. Records and sealed classes and interfaces are discussed [Chapter 17](#).

Before moving on, it is useful to mention another **Class** capability that you may find interesting. Beginning with JDK 11, **Class** provides three methods that relate to a nest. A *nest* is a group of classes and/or interfaces nested within an outer class or interface. The nest concept enables the JVM to more efficiently handle certain situations involving access between nest members. It is important to state that a nest is *not* a source code mechanism, and it does *not* change the Java language or how it defines accessibility. Nests relate specifically to how the compiler and JVM work. However, it is now possible to obtain a nest's top-level class/interface, which is called the *nest host*, by use of `getNestHost()`. You can determine if one class/interface is a member of the same nest as another by use of `isNestMateOf()`. Finally, you can get an array containing a list of the nest members by calling `getNestMembers()`. You may find these methods useful when using reflection, for example.

ClassLoader

The abstract class **ClassLoader** defines how classes are loaded. Your application can create subclasses that extend **ClassLoader**, implementing its methods. Doing so allows you to load classes in some way other than the way they are normally loaded by the Java run-time system. However, this is not something that you will normally need to do.

Math

The **Math** class contains all the floating-point functions that are used for geometry and trigonometry, as well as several general-purpose methods. **Math** defines three well-known **double** constants: **E** (approximately 2.72), **PI** (approximately 3.14), and **TAU** (approximately 6.28).

Trigonometric Functions

The following methods accept a **double** parameter for an angle in radians and return the result of their respective trigonometric function:

Method	Description
static double sin(double <i>arg</i>)	Returns the sine of the angle specified by <i>arg</i> in radians
static double cos(double <i>arg</i>)	Returns the cosine of the angle specified by <i>arg</i> in radians
static double tan(double <i>arg</i>)	Returns the tangent of the angle specified by <i>arg</i> in radians

The next methods take as a parameter the result of a trigonometric function and return, in radians, the angle that would produce that result. They are the inverse of their non-arc companions.

Method	Description
static double asin(double <i>arg</i>)	Returns the angle whose sine is specified by <i>arg</i>
static double acos(double <i>arg</i>)	Returns the angle whose cosine is specified by <i>arg</i>
static double atan(double <i>arg</i>)	Returns the angle whose tangent is specified by <i>arg</i>
static double atan2(double <i>x</i> , double <i>y</i>)	Returns the angle whose tangent is <i>x/y</i>

The next methods compute the hyperbolic sine, cosine, and tangent of an angle:

Method	Description
static double sinh(double <i>arg</i>)	Returns the hyperbolic sine of the angle specified by <i>arg</i>
static double cosh(double <i>arg</i>)	Returns the hyperbolic cosine of the angle specified by <i>arg</i>
static double tanh(double <i>arg</i>)	Returns the hyperbolic tangent of the angle specified by <i>arg</i>

Exponential Functions

Math defines the following exponential methods:

Method	Description
static double cbrt(double <i>arg</i>)	Returns the cube root of <i>arg</i>
static double exp(double <i>arg</i>)	Returns e to the <i>arg</i>
static double expm1(double <i>arg</i>)	Returns e to the <i>arg</i> -1
static double log(double <i>arg</i>)	Returns the natural logarithm of <i>arg</i>
static double log10(double <i>arg</i>)	Returns the base 10 logarithm for <i>arg</i>
static double log1p(double <i>arg</i>)	Returns the natural logarithm for <i>arg</i> + 1
static double pow(double <i>y</i> , double <i>x</i>)	Returns <i>y</i> raised to the <i>x</i> ; for example, pow(2.0, 3.0) returns 8.0
static double scalb(double <i>arg</i> , int <i>factor</i>)	Returns <i>arg</i> × 2^{factor}
static float scalb(float <i>arg</i> , int <i>factor</i>)	Returns <i>arg</i> × 2^{factor}
static double sqrt(double <i>arg</i>)	Returns the square root of <i>arg</i>

Rounding Functions

The **Math** class defines several methods that provide various types of rounding operations. They are shown in [Table 19-16](#). Notice the two **ulp()** methods at the end of the table. In this context, *ulp* stands for *units in the last place*. It indicates the distance between a value and the next higher value. It can be used to help assess the accuracy of a result.

Method	Description
static int abs(int <i>arg</i>)	Returns the absolute value of <i>arg</i> .
static long abs(long <i>arg</i>)	Returns the absolute value of <i>arg</i> .
static float abs(float <i>arg</i>)	Returns the absolute value of <i>arg</i> .
static double abs(double <i>arg</i>)	Returns the absolute value of <i>arg</i> .
static double ceil(double <i>arg</i>)	Returns the ceiling of <i>arg</i> , that is, the smallest whole number greater than or equal to <i>arg</i> .
static double floor(double <i>arg</i>)	Returns the floor of <i>arg</i> , that is, the largest whole number less than or equal to <i>arg</i> .
static int ceilDiv(int <i>dividend</i> , int <i>divisor</i>)	Returns the ceiling of the result of <i>dividend</i> / <i>divisor</i> .
static int ceilDiv(long <i>dividend</i> , int <i>divisor</i>)	Returns the ceiling of the result of <i>dividend</i> / <i>divisor</i> .
static long ceilDiv(long <i>dividend</i> , long <i>divisor</i>)	Returns the ceiling of the result of <i>dividend</i> / <i>divisor</i> .
static int ceilDivExact(int <i>dividend</i> , int <i>divisor</i>)	Returns the ceiling of result of <i>dividend</i> / <i>divisor</i> . Throws an ArithmeticException if an overflow occurs.
static long ceilDivExact(long <i>dividend</i> , long <i>divisor</i>)	Returns the ceiling of the result of <i>dividend</i> / <i>divisor</i> . Throws an ArithmeticException if an overflow occurs.
static int ceilMod(int <i>dividend</i> , int <i>divisor</i>)	Returns the ceiling of the remainder of <i>dividend</i> / <i>divisor</i> .
static int ceilMod(long <i>dividend</i> , int <i>divisor</i>)	Returns the ceiling of the remainder of <i>dividend</i> / <i>divisor</i> .
static long ceilMod(long <i>dividend</i> , long <i>divisor</i>)	Returns the ceiling of the remainder of <i>dividend</i> / <i>divisor</i> .
static int divideExact(int <i>dividend</i> , int <i>divisor</i>)	Returns the result of <i>dividend</i> / <i>divisor</i> . Throws an ArithmeticException if an overflow occurs.
static long divideExact(long <i>dividend</i> , long <i>divisor</i>)	Returns the result of <i>dividend</i> / <i>divisor</i> . Throws an ArithmeticException if an overflow occurs.
static double clamp(double <i>value</i> ,	If <i>value</i> is less than <i>min</i> , then return <i>min</i> ;

<code>static float clamp(float value, float min, float max)</code>	If <i>value</i> is less than <i>min</i> , then return <i>min</i> ; if greater than <i>max</i> , return <i>max</i> . Otherwise, return <i>value</i> .
<code>static int clamp(long value, int min, int max)</code>	If <i>value</i> is less than <i>min</i> , then return <i>min</i> ; if greater than <i>max</i> , return <i>max</i> . Otherwise, return <i>value</i> .
<code>static long clamp(long value, long min, long max)</code>	If <i>value</i> is less than <i>min</i> , then return <i>min</i> ; if greater than <i>max</i> , return <i>max</i> . Otherwise, return <i>value</i> .
<code>static int floorDiv(int dividend, int divisor)</code>	Returns the floor of the result of <i>dividend/divisor</i> .
<code>static long floorDiv(long dividend, int divisor)</code>	Returns the floor of the result of <i>dividend/divisor</i> .
<code>static long floorDiv(long dividend, long divisor)</code>	Returns the floor of the result of <i>dividend/divisor</i> .
<code>static int floorDivExact(int dividend, int divisor)</code>	Returns the floor of the result of <i>dividend/divisor</i> . Throws an ArithmeticException if an overflow occurs.
<code>static long floorDivExact(long dividend, long divisor)</code>	Returns the floor of the result of <i>dividend/divisor</i> . Throws an ArithmeticException if an overflow occurs.
<code>static int floorMod(int dividend, int divisor)</code>	Returns the floor of the remainder of <i>dividend/divisor</i> .
<code>static int floorMod(long dividend, int divisor)</code>	Returns the floor of the remainder of <i>dividend/divisor</i> .
<code>static long floorMod(long dividend, long divisor)</code>	Returns the floor of the remainder of <i>dividend/divisor</i> .
<code>static int max(int x, int y)</code>	Returns the maximum of <i>x</i> and <i>y</i> .
<code>static long max(long x, long y)</code>	Returns the maximum of <i>x</i> and <i>y</i> .
<code>static float max(float x, float y)</code>	Returns the maximum of <i>x</i> and <i>y</i> .
<code>static double max(double x, double y)</code>	Returns the maximum of <i>x</i> and <i>y</i> .
<code>static int min(int x, int y)</code>	Returns the minimum of <i>x</i> and <i>y</i> .
<code>static long min(long x, long y)</code>	Returns the minimum of <i>x</i> and <i>y</i> .
<code>static float min(float x, float y)</code>	Returns the minimum of <i>x</i> and <i>y</i> .

static double min(double <i>x</i> , double <i>y</i>)	Returns the minimum of <i>x</i> and <i>y</i> .
static double nextAfter(double <i>arg</i> , double <i>toward</i>)	Beginning with the value of <i>arg</i> , returns the next value in the direction of <i>toward</i> . If <i>arg</i> == <i>toward</i> , then <i>toward</i> is returned.
static float nextAfter(float <i>arg</i> , double <i>toward</i>)	Beginning with the value of <i>arg</i> , returns the next value in the direction of <i>toward</i> . If <i>arg</i> == <i>toward</i> , then <i>toward</i> is returned.
static double nextDown(double <i>val</i>)	Returns the next value lower than <i>val</i> .

Table 19-16 The Rounding Methods Defined by **Math**

Miscellaneous Math Methods

In addition to the methods just shown, **Math** defines several other methods, which are shown in [Table 19-17](#). Notice that several of the methods use the suffix **Exact**. They throw an **ArithmeticException** if overflow occurs. Thus, these methods give you an easy way to watch various operations for overflow.

Method	Description
static int absExact(int <i>arg</i>)	Returns the absolute value of <i>arg</i> .
static long absExact(long <i>arg</i>)	Returns the absolute value of <i>arg</i> .
static int addExact(int <i>arg1</i> , int <i>arg2</i>)	Returns <i>arg1</i> + <i>arg2</i> . Throws an ArithmaticException if overflow occurs.
static long addExact(long <i>arg1</i> , long <i>arg2</i>)	Returns <i>arg1</i> + <i>arg2</i> . Throws an ArithmaticException if overflow occurs.
static double copySign(double <i>arg</i> , double <i>signarg</i>)	Returns <i>arg</i> with same sign as that of <i>signarg</i> .
static float copySign(float <i>arg</i> , float <i>signarg</i>)	Returns <i>arg</i> with same sign as that of <i>signarg</i> .
static int decrementExact(int <i>arg</i>)	Returns <i>arg</i> - 1. Throws an ArithmaticException if overflow occurs.
static long decrementExact(long <i>arg</i>)	Returns <i>arg</i> - 1. Throws an ArithmaticException if overflow occurs.
static double fma(double <i>arg1</i> , double <i>arg2</i> , double <i>arg3</i>)	Adds <i>arg3</i> to the product of <i>arg1</i> and <i>arg2</i> and returns the rounded result. The name is short for fused multiply add.
static float fma(float <i>arg1</i> , float <i>arg2</i> , float <i>arg3</i>)	Adds <i>arg3</i> to the product of <i>arg1</i> and <i>arg2</i> and returns the rounded result. The name is short for fused multiply add.
static int getExponent(double <i>arg</i>)	Returns the base-2 exponent used by the binary representation of <i>arg</i> .
static int getExponent(float <i>arg</i>)	Returns the base-2 exponent used by the binary representation of <i>arg</i> .
static hypot(double <i>side1</i> , double <i>side2</i>)	Returns the length of the hypotenuse of a right triangle given the length of the two opposing sides.

static double IEEEremainder(double <i>dividend</i> , double <i>divisor</i>)	Returns the remainder of <i>dividend</i> / <i>divisor</i> .
static int incrementExact(int <i>arg</i>)	Returns <i>arg</i> + 1. Throws an ArithmeticException if overflow occurs.
static long incrementExact(long <i>arg</i>)	Returns <i>arg</i> + 1. Throws an ArithmeticException if overflow occurs.
static int multiplyExact(int <i>arg1</i> , int <i>arg2</i>)	Returns <i>arg1</i> * <i>arg2</i> . Throws an ArithmeticException if overflow occurs.
static long multiplyExact(long <i>arg1</i> , int <i>arg2</i>)	Returns <i>arg1</i> * <i>arg2</i> . Throws an ArithmeticException if overflow occurs.
static long multiplyExact(long <i>arg1</i> , long <i>arg2</i>)	Returns <i>arg1</i> * <i>arg2</i> . Throws an ArithmeticException if overflow occurs.
static long multiplyFull(int <i>arg1</i> , int <i>arg2</i>)	Returns <i>arg1</i> * <i>arg2</i> as a long value.
static long multiplyHigh(long <i>arg1</i> , long <i>arg2</i>)	Returns a long value that contains the most significant bits of <i>arg1</i> * <i>arg2</i> .
static int negateExact(int <i>arg</i>)	Returns - <i>arg</i> . Throws an ArithmeticException if overflow occurs.
static long negateExact(long <i>arg</i>)	Returns - <i>arg</i> . Throws an ArithmeticException if overflow occurs.
static double random()	Returns a pseudorandom number between 0 and 1.
static float signum(double <i>arg</i>)	Determines the sign of a value. It returns 0 if <i>arg</i> is 0, 1 if <i>arg</i> is greater than 0, and -1 if <i>arg</i> is less than 0.
static float signum(float <i>arg</i>)	Determines the sign of a value. It returns 0 if <i>arg</i> is 0, 1 if <i>arg</i> is greater than 0, and -1 if <i>arg</i> is less than 0.
static int subtractExact(int <i>arg1</i> , int <i>arg2</i>)	Returns <i>arg1</i> - <i>arg2</i> . Throws an ArithmeticException if overflow occurs.
static long subtractExact(long <i>arg1</i> , long <i>arg2</i>)	Returns <i>arg1</i> - <i>arg2</i> . Throws an ArithmeticException if overflow occurs.
static double toDegrees(double <i>angle</i>)	Converts radians to degrees. The angle passed must be specified in radians. The result in degrees is returned.
static int toIntExact(long <i>arg</i>)	Returns <i>arg</i> as an int. Throws an ArithmeticException if overflow occurs.
static double toRadians(double <i>angle</i>)	Converts degrees to radians. The <i>angle</i> passed must be specified in degrees. The result in radians is returned.
static long unsignedMultiplyHigh(int <i>arg1</i> , int <i>arg2</i>)	Returns <i>arg1</i> * <i>arg2</i> as an unsigned 64-bit long.

Table 19-17 Other Methods Defined by **Math**

The following program demonstrates **toRadians()** and **toDegrees()**:

The output is shown here:

StrictMath

The **StrictMath** class defines a complete set of mathematical methods that parallel those in **Math**. The difference is that the **StrictMath** version is guaranteed to generate precisely identical results across all Java implementations, whereas the methods in **Math** are given more latitude in order to improve performance. It is important to point out that beginning with JDK 17, all math computations are now strict.

Compiler

The **Compiler** class supports the creation of Java environments in which Java bytecode is compiled into executable code rather than interpreted. It is not for normal programming use and has been deprecated for removal.

Thread, ThreadGroup, and Runnable

The **Runnable** interface and the **Thread** and **ThreadGroup** classes support multithreaded programming. Each is examined next.

NOTE An overview of the techniques used to manage threads, implement the **Runnable** interface, and create multithreaded programs is presented in [Chapter 11](#).

The Runnable Interface

The **Runnable** interface must be implemented by any class that will initiate a separate thread of execution. **Runnable** only defines one abstract method, called **run()**, which is the entry point to the thread. It is defined like this:

```
void run( )
```

Threads that you create must implement this method.

Thread

Thread creates a new thread of execution. It implements **Runnable** and defines a number of constructors. Several are shown here:

threadOb is an instance of a class that implements the **Runnable** interface and defines where execution of the thread will begin. The name of the thread is specified by *threadName*. When a name is not specified, one is created by the Java Virtual Machine. *groupOb* specifies the thread group to which the new thread will belong. When no thread group is specified, by default the new thread belongs to the same group as the parent thread.

The following constants are defined by **Thread**:

MAX_PRIORITY
MIN_PRIORITY
NORM_PRIORITY

As expected, these constants specify the maximum, minimum, and default thread priorities, respectively.

The non-deprecated methods defined by **Thread** are shown in [Table 19-18](#). **Thread** also includes the deprecated methods **stop()**, **suspend()**, and **resume()**. However, as explained in [Chapter 11](#), these were deprecated because they were inherently unstable. Also deprecated are **countStackFrames()**, because it calls **suspend()**, and **destroy()**, because it can cause deadlock. Furthermore, beginning with JDK 11, **destroy()** and one version of **stop()** have now been removed from **Thread**. Also, JDK 17 deprecated **checkAccess()** for removal.

Method	Description
static int activeCount()	Returns the approximate number of active threads in the group to which the thread belongs.
static Thread currentThread()	Returns a Thread object that encapsulates the thread that calls this method.
static void dumpStack()	Displays the call stack for the thread.
static int enumerate(Thread[] <i>threads</i>)	Puts copies of all Thread objects in the current thread's group into <i>threads</i> . The number of threads is returned.
static Map<Thread, StackTraceElement[]> getAllStackTraces()	Returns a Map that contains the stack traces for all live platform threads. In the map, each entry consists of a key, which is the Thread object, and its value, which is an array of StackTraceElement .
ClassLoader getContextClassLoader()	Returns the context class loader that is used to load classes and resources for this thread.
static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()	Returns the default uncaught exception handler.
final String getName()	Returns the thread's name.
final int getPriority()	Returns the thread's priority setting.
StackTraceElement[] getStackTrace()	Returns an array containing the stack trace for the invoking thread.
Thread.State getState()	Returns the invoking thread's state.
final ThreadGroup getThreadGroup()	Returns the ThreadGroup object of which the invoking thread is a member.

<code>Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()</code>	Returns the invoking thread's uncaught exception handler.
<code>static boolean holdsLock(Object ob)</code>	Returns true if the invoking thread owns the lock on <i>ob</i> . Returns false otherwise.
<code>void interrupt()</code>	Interrupts the thread.
<code>static boolean interrupted()</code>	Returns true if the currently executing thread has been interrupted. Otherwise, it returns false .
<code>final boolean isAlive()</code>	Returns true if the thread is still active. Otherwise, it returns false .
<code>final boolean isDaemon()</code>	Returns true if the thread is a daemon thread. Otherwise, it returns false .
<code>boolean isInterrupted()</code>	Returns true if the invoking thread has been interrupted. Otherwise, it returns false .
<code>public final boolean isVirtual()</code>	Returns true if the thread is virtual, that is, if it is scheduled by the JVM rather than the operating system.
<code>final void join() throws InterruptedException</code>	Waits until the thread terminates.
<code>final boolean join(Duration duration) throws InterruptedException</code>	Waits for the time defined by the <i>duration</i> object for the thread to terminate, returning true if the thread terminated.
<code>final void join(long milliseconds) throws InterruptedException</code>	Waits up to the specified number of milliseconds for the thread on which it is called to terminate.
<code>final void join(long milliseconds, int nanoseconds) throws InterruptedException</code>	Waits up to the specified number of milliseconds plus nanoseconds for the thread on which it is called to terminate.
<code>static Thread.Builder.OfPlatform ofPlatform()</code>	Returns a builder for creating platform threads, i.e., those scheduled by the operating system.
<code>static Thread.Builder.OfVirtual ofVirtual()</code>	Returns a builder for creating virtual threads, i.e., threads scheduled by Java not the operating system.
<code>static void onSpinWait()</code>	Called to signify that execution is currently inside a wait loop, possibly enabling a runtime optimization.
<code>void run()</code>	Begins execution of a thread.

<code>void setContextClassLoader(ClassLoader <i>cl</i>)</code>	Sets the context class loader that will be used by the invoking thread to <i>cl</i> .
<code>final void setDaemon(boolean <i>state</i>)</code>	Flags the thread as a daemon thread.
<code>static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler <i>e</i>)</code>	Sets the default uncaught exception handler to <i>e</i> .

Table 19-18 The Non-Degraded Methods Defined by **Thread**

ThreadGroup

ThreadGroup creates a group of threads. It defines these two constructors:

For both forms, *groupName* specifies the name of the thread group. The first version creates a new group that has the current thread as its parent. In

the second form, the parent is specified by *parentOb*. The non-deprecated methods defined by **ThreadGroup** are shown in [Table 19-19](#).

Method	Description
int activeCount()	Returns the approximate number of active threads in the invoking group (including those in subgroups).
int activeGroupCount()	Returns the approximate number of active groups (including subgroups) for which the invoking thread is a parent.

Table 19-19 The Non-Degraded Methods Defined by **ThreadGroup**

Thread groups offer a convenient way to manage groups of threads as a unit. This is particularly valuable in situations in which you want to suspend and resume a number of related threads. For example, imagine a program in which one set of threads is used for printing a document, another set is used to display the document on the screen, and another set saves the document to a disk file. If printing is aborted, you will want an easy way to stop all threads related to printing. Thread groups offer this convenience. The following program, which creates two thread groups of two threads each, illustrates this usage:

Sample output from this program is shown here (the precise output you see may differ):

Inside the program, notice that thread group A is suspended for four seconds. As the output confirms, this causes threads One and Two to pause, but threads Three and Four continue running. After the four seconds, threads One and Two are resumed. Notice how thread group A is suspended and resumed. First, the threads in group A are obtained by calling **enumerate()** on group A. Then, each thread is suspended by iterating through the resulting array. To resume the threads in A, the list is again traversed and each thread is resumed.

ThreadLocal and InheritableThreadLocal

Java defines two additional thread-related classes in **java.lang**:

- **ThreadLocal** Used to create thread local variables. Each thread will have its own copy of a thread local variable.
- **InheritableThreadLocal** Creates thread local variables that may be inherited.

Package

Package encapsulates information about a package. The methods defined by **Package** are shown in [Table 19-20](#). The following program demonstrates **Package**, displaying the packages about which the program currently is aware:

Method	Description
<A extends Annotation> A getAnnotation(Class<A> annoType)	Returns an Annotation object that contains the annotation associated with <i>annoType</i> for the invoking object.
Annotation[] getAnnotations()	Returns all annotations associated with the invoking object in an array of Annotation objects. Returns a reference to this array.
<A extends Annotation> A[] getAnnotationsByType(Class<A> annoType)	Returns an array of the annotations (including repeated annotations) of <i>annoType</i> associated with the invoking object.
<A extends Annotation> A getDeclaredAnnotation(Class<A> annoType)	Returns an Annotation object that contains the non-inherited annotation associated with <i>annoType</i> .
Annotation[] getDeclaredAnnotations()	Returns an Annotation object for all the annotations that are declared by the invoking object. (Inherited annotations are ignored.)
<A extends Annotation> A[] getDeclaredAnnotationsByType(Class<A> annoType)	Returns an array of the non-inherited annotations (including repeated annotations) of <i>annoType</i> associated with the invoking object.
String getImplementationTitle()	Returns the title of the invoking package.
String getImplementationVendor()	Returns the name of the implementor of the invoking package.
String getImplementationVersion()	Returns the version number of the invoking package.
String getName()	Returns the name of the invoking package.
static Package[] getPackages()	Returns all packages about which the invoking program is currently aware.
String getSpecificationTitle()	Returns the title of the invoking package's specification.
String getSpecificationVendor()	Returns the name of the owner of the specification for the invoking package.
String getSpecificationVersion()	Returns the invoking package's specification version number.
int hashCode()	Returns the hash code for the invoking package.
boolean isAnnotationPresent(Class<? extends Annotation> anno)	Returns true if the annotation described by <i>anno</i> is associated with the invoking object. Returns false otherwise.

Table 19-20 The Methods Defined by **Package**

<code>boolean isCompatibleWith(String verNum)</code> throws NumberFormatException	Returns true if <i>verNum</i> is less than or equal to the invoking package's version number.
<code>boolean isSealed()</code>	Returns true if the invoking package is sealed. Returns false otherwise.
<code>boolean isSealed(URL url)</code>	Returns true if the invoking package is sealed relative to <i>url</i> . Returns false otherwise.
<code>String toString()</code>	Returns the string equivalent of the invoking package.

Module

Added by JDK 9, the **Module** class encapsulates a module. Using a **Module** instance you can add various access rights to a module, determine access rights, or obtain information about a module. For example, to export a package to a specified module, call **addExports()**; to open a package to a specified module, call **addOpens()**; to read another module, call **addReads()**; and to add a service requirement, call **addUses()**. You can determine if a module can access another by calling **canRead()**. To determine if a module uses a service, call **canUse()**. Although these methods will be most useful in specialized situations, **Module** defines several others that may be of more general interest.

For example, you can obtain the name of a module by calling **getName()**. If called from within a named module, the name is returned. If called from the unnamed module, **null** is returned. You can obtain a **Set** of the packages in a module by calling **getPackages()**. A module descriptor, in the form of a **ModuleDescriptor** instance, is returned by **getDescriptor()**. (**ModuleDescriptor** is a class declared in **java.lang.module**.) You can determine if a package is exported or opened by the invoking module by

calling `isExported()` or `isOpen()`. Use `isNamed()` to determine if a module is named or unnamed. Other methods include `getAnnotation()`, `getDeclaredAnnotations()`, `getLayer()`, `getClassLoader()`, and `getResourceAsStream()`. The `toString()` method is also overridden for **Module**.

Assuming the modules defined by the examples in [Chapter 16](#), you can easily experiment with the **Module** class. For example, try adding the following lines to the **MyModAppDemo** class:

Here, the methods `getName()` and `getPackages()` are used. Notice that a **Module** instance is obtained by calling `getModule()` on the **Class** instance for **MyModAppDemo**. When run, these lines produce the following output:

ModuleLayer

ModuleLayer, added by JDK 9, encapsulates a module layer. The nested class **ModuleLayer.Controller**, also added by JDK 9, is the controller for a module layer. In general, these classes are for specialized applications.

RuntimePermission

RuntimePermission relates to Java's security mechanism.

Throwable

The **Throwable** class supports Java's exception-handling system and is the class from which all exception classes are derived. It is discussed in [Chapter 10](#).

SecurityManager

SecurityManager has been deprecated for removal by JDK 17. Consult the Java documentation for the latest details.

StackTraceElement

The **StackTraceElement** class describes a single *stack frame*, which is an individual element of a stack trace when an exception occurs. Each stack frame represents an *execution point*, which includes such things as the name of the class, the name of the method, the name of the file, and the source-code line number. Beginning with JDK 9, module information is also included. **StackTraceElement** defines two constructors, but typically you won't need to use them because an array of **StackTraceElements** is returned by various methods, such as the **getStackTrace()** method of the **Throwable** and **Thread** classes.

The methods supported by **StackTraceElement** are shown in [Table 19-21](#). These methods give you programmatic access to a stack trace.

Method	Description
boolean equals(Object <i>ob</i>)	Returns true if the invoking StackTraceElement is the same as the one passed in <i>ob</i> . Otherwise, it returns false .
String getClassLoaderName()	Returns the name of the class loader used to load the class in which the execution point described by the invoking StackTraceElement occurred. If the object does not include class loader information, null is returned.
String getClassName()	Returns the name of the class in which the execution point described by the invoking StackTraceElement occurred.
String getFileName()	Returns the name of the file in which the source code of the execution point described by the invoking StackTraceElement is stored.
int getLineNumber()	Returns the source-code line number at which the execution point described by the invoking StackTraceElement occurred. In some situations, the line number will not be available, in which case a negative value is returned.
String getMethodName()	Returns the name of the method in which the execution point described by the invoking StackTraceElement occurred.
String getModuleName()	Returns the name of the module in which the execution point described by the invoking StackTraceElement occurred. If the object does not include module information, null is returned.

Table 19-21 The Methods Defined by **StackTraceElement**

StackWalker and StackWalker.StackFrame

Added by JDK 9, the **StackWalker** class and the **StackWalker.StackFrame** interface support stack walking operations. A **StackWalker** instance is obtained by use of the static **getInstance()** method defined by **StackWalker**. Stack walking is initiated by calling the **walk()** method of **StackWalker**. Each stack frame is encapsulated as a **StackWalker.StackFrame** object. The **StackWalker.Option** enumeration was also added.

Enum

As described in [Chapter 12](#), an enumeration is a list of named constants. (Recall that an enumeration is created by using the keyword **enum**.) All enumerations automatically inherit **Enum**. **Enum** is a generic class that is declared as shown here:

```
class Enum<E extends Enum<E>>
```

Here, **E** stands for the enumeration type. **Enum** has no public constructors.

Enum defines several commonly used methods, which are shown in [Table 19-22](#). Beginning with JDK 12, **Enum** also implements the **Constable** interface, which specifies the **describeConstable()** method. JDK 12 also added the **Enum.EnumDesc** class.

Method	Description
<code>protected final Object clone() throws CloneNotSupportedException</code>	Invoking this method causes a CloneNotSupportedException to be thrown. This prevents enumerations from being cloned.
<code>final int compareTo(E e)</code>	Compares the ordinal value of two constants of the same enumeration. Returns a negative value if the invoking constant has an ordinal value less than e's, zero if the two ordinal values are the same, and a positive value if the invoking constant has an ordinal value greater than e's.

Table 19-22 Commonly Used Methods Defined by **Enum**

Record

Added by JDK 16, **Record** is the superclass for all records. In other words, all records automatically inherit **Record**. It defines no methods of its own, but overrides **equals()**, **hashCode()**, and **toString()**, which are inherited from **Object**. Records are discussed in [Chapter 17](#).

ClassValue

ClassValue can be used to associate a value with a type. It is a generic class defined like this:

```
Class ClassValue<T>
```

It is designed for highly specialized uses, not for normal programming.

The CharSequence Interface

The **CharSequence** interface defines methods that grant read-only access to a sequence of characters. These methods are shown in [Table 19-23](#). This interface is implemented by **String**, **StringBuffer**, and **StringBuilder**, among others.

Method	Description
char charAt(int <i>idx</i>)	Returns the character at the index specified by <i>idx</i> .
static int compare(CharSequence <i>seqA</i> , CharSequence <i>seqB</i>)	Compares <i>seqA</i> to <i>seqB</i> . Returns 0 if the sequences are the same. Returns a negative value if <i>seqA</i> is less than <i>seqB</i> . Returns a positive value if <i>seqA</i> is greater than <i>seqB</i> .

Table 19-23 The Methods Defined by **CharSequence**

The Comparable Interface

Objects of classes that implement **Comparable** can be ordered. In other words, classes that implement **Comparable** contain objects that can be compared in some meaningful manner. **Comparable** is generic and is declared like this:

```
interface Comparable<T>
```

Here, **T** represents the type of objects being compared.

The **Comparable** interface declares one method that is used to determine what Java calls the *natural ordering* of instances of a class. The signature of the method is shown here:

```
int compareTo(T obj)
```

This method compares the invoking object with *obj*. It returns 0 if the values are equal. A negative value is returned if the invoking object has a lower value. Otherwise, a positive value is returned.

This interface is implemented by several of the classes already reviewed in this book, such as **Byte**, **Character**, **Double**, **Float**, **Long**, **Short**, **String**, **Integer**, and **Enum**.

The Appendable Interface

An object of a class that implements **Appendable** can have a character or character sequences appended to it. **Appendable** defines these three methods:

In the first form, the character *ch* is appended to the invoking object. In the second form, the character sequence *chars* is appended to the invoking object. The third form allows you to indicate a portion (the characters running from *begin* through *end*-1) of the sequence specified by *chars*. In all cases, a reference to the invoking object is returned.

The Iterable Interface

Iterable must be implemented by any class whose objects will be used by the for-each version of the **for** loop. In other words, in order for an object to be used within a for-each style **for** loop, its class must implement **Iterable**. **Iterable** is a generic interface that has this declaration:

```
interface Iterable<T>
```

Here, **T** is the type of the object being iterated. It defines one abstract method, **iterator()**, which is shown here:

```
Iterator<T> iterator( )
```

It returns an iterator to the elements contained in the invoking object.

Iterable also defines two default methods. The first is called **forEach()**:

```
default void forEach(Consumer<? super T> action)
```

For each element being iterated, **forEach()** executes the code specified by *action*. (**Consumer** is a functional interface defined in **java.util.function**.

See [Chapter 21](#).)

The second default method is **spliterator()**, shown next:

```
default Spliterator<T> spliterator( )
```

It returns a **Spliterator** to the sequence being iterated. (See [Chapters 20](#) and [30](#) for details on spliterators.)

NOTE Iterators are described in detail in [Chapter 20](#).

The Readable Interface

The **Readable** interface indicates that an object can be used as a source for characters. It defines one method called **read()**, which is shown here:

```
int read(CharBuffer buf) throws IOException
```

This method reads characters into *buf*. It returns the number of characters read, or `-1` if an EOF is encountered.

The AutoCloseable Interface

AutoCloseable provides support for the **try-with-resources** statement, which implements what is sometimes referred to as *automatic resource management* (ARM). The **try-with-resources** statement automates the process of releasing a resource (such as a stream) when it is no longer needed. (See [Chapter 13](#) for details.) Only objects of classes that implement **AutoCloseable** can be used with **try-with-resources**. The **AutoCloseable** interface defines only the **close()** method, which is shown here:

```
void close() throws Exception
```

This method closes the invoking object, releasing any resources that it may hold. It is automatically called at the end of a **try-with-resources** statement, thus eliminating the need to explicitly invoke **close()**. **AutoCloseable** is implemented by several classes, including all of the I/O classes that open a stream that can be closed.

The Thread.UncaughtExceptionHandler Interface

The static **Thread.UncaughtExceptionHandler** interface is implemented by classes that want to handle uncaught exceptions. It is implemented by **ThreadGroup**. It declares only one method, which is shown here:

```
void uncaughtException(Thread thrd, Throwable exc)
```

Here, *thrd* is a reference to the thread that generated the exception and *exc* is a reference to the exception.

The **java.lang** Subpackages

Java defines several subpackages of **java.lang**. Except as otherwise noted, these packages are in the **java.base** module.

- `java.lang.annotation`

- `java.lang.constant`
- `java.lang.instrument`
- `java.lang.invoke`
- `java.lang.management`
- `java.lang.module`
- `java.lang.ref`
- `java.lang.reflect`

Each is briefly described here.

java.lang.annotation

Java's annotation facility is supported by `java.lang.annotation`. It defines the **Annotation** interface, the **ElementType** and **RetentionPolicy** enumerations, and several predefined annotations. Annotations are described in [Chapter 12](#).

java.lang.constant

`java.lang.constant` is a specialized package that supports descriptors for constants. It is typically used by applications that access bytecode, and it was added by JDK 12.

java.lang.instrument

`java.lang.instrument` defines features that can be used to add instrumentation to various aspects of program execution. It defines the **Instrumentation** and **ClassFileTransformer** interfaces, and the **ClassDefinition** class. This package is in the **java.instrument** module.

java.lang.invoke

`java.lang.invoke` supports dynamic language features. It includes classes such as **CallSite**, **MethodHandle**, and **MethodType**.

java.lang.management

The **java.lang.management** package provides management support for the JVM and the execution environment. Using the features in **java.lang.management**, you can observe and manage various aspects of program execution. This package is in the **java.management** module.

java.lang.module

The **java.lang.module** package supports modules. It includes classes such as **ModuleDescriptor** and **ModuleReference**, and the interfaces **ModuleFinder** and **ModuleReader**.

java.lang.ref

You learned earlier that the garbage collection facilities in Java automatically determine when no references exist to an object. The object is then assumed to be no longer needed and its memory is reclaimed. The classes in the **java.lang.ref** package provide more flexible control over the garbage collection process.

java.lang.reflect

Reflection is the ability of a program to analyze code at run time. The **java.lang.reflect** package provides the ability to obtain information about the fields, constructors, methods, and modifiers of a class. Among other reasons, you need this information to build software tools that enable you to work with Java Beans components. The tools use reflection to determine dynamically the characteristics of a component. Reflection was introduced in [Chapter 12](#) and is also examined in [Chapter 31](#).

java.lang.reflect defines several classes, including **Method**, **Field**, and **Constructor**. It also defines several interfaces, including **AnnotatedElement**, **Member**, and **Type**. In addition, the **java.lang.reflect** package includes the **Array** class, which enables you to create and access arrays dynamically.