JAVA PROGRAMMING

FROM PROBLEM ANALYSIS TO PROGRAM DESIGN

5 EDITION

D.S. Malik

Java Programming

From Problem Analysis to Program Design

Fifth Edition

D.S. Malik

COURSE TECHNOLOGY
CENGAGE Learning™

Australia • Brazil • Japan • Korea • Mexico • Singapore • Spain • United Kingdom • United States

COURSE TECHNOLOGY
CENGAGE Learning™

Java Programming: From Problem Analysis to Program Design, Fifth Edition
D.S. Malik

Executive Editor: Marie Lee
Acquisitions Editor: Brandi Shailer
Senior Product Manager: Alyssa Pratt
Editorial Assistant: Jacqueline Lacaire
Content Project Manager: Lisa Weidenfeld
Associate Marketing Manager: Shanna Shelton
Art Director: Faith Brosnan
Proofreader: Andrea Schein
Indexer: Alexandra Nickerson
Print Buyer: Julio Esperas
Cover Designer: Roycroft Design/ www.roycroftdesign.com
Cover Photo: © photolibrary/Richard Cummins
Compositor: Integra

Library of Congress Control Number: 2010940363

ISBN-13: 978-1-111-53053-2

ISBN-10: 1-111-53053-x

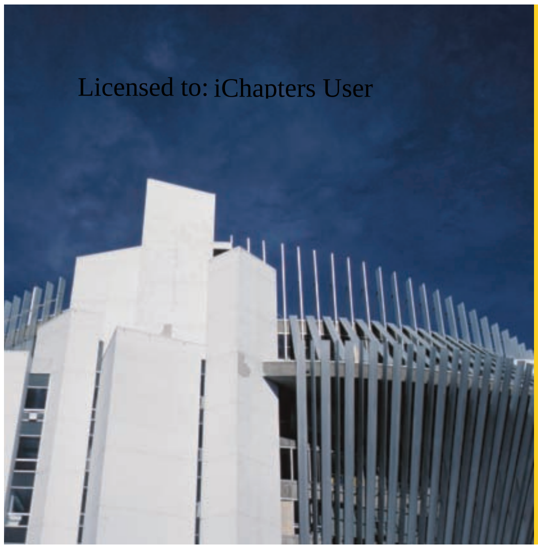
Course Technology
20 Channel Center Street
Boston, MA 02210
USA

Printed in the United States of America
1 2 3 4 5 6 7 16 15 14 13 12 11 10

TO

My Daughter

Shelly Malik



Brief Contents

PREFACE xix

1. An Overview of Computers and Programming Languages 1

2. Basic Elements of Java 25

3. Introduction to Objects and Input/Output 113

4. Control Structures I: Selection 177

5. Control Structures II: Repetition 249

6. Graphical User Interface (GUI) and Object-Oriented Design (OOD) 327

7. User-Defined Methods 383

8. User-Defined Classes and ADTs 465

9. Arrays 551

10. Inheritance and Polymorphism 639

11. Handling Exceptions and Events 723

12. Advanced GUIs and Graphics 783

13. Recursion 873

14. Searching and Sorting 907

APPENDIX A Java Reserved Words 939

APPENDIX B Operator Precedence 941

APPENDIX C Character Sets 945

APPENDIX D Additional Java Topics 949

APPENDIX E Answers to Odd-Numbered Exercises 997

INDEX 1023

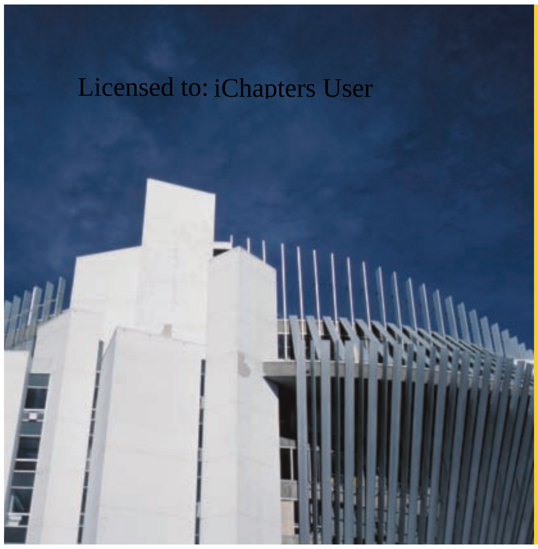


Table of Contents

Preface xix

1 AN OVERVIEW OF COMPUTERS AND PROGRAMMING LANGUAGES 1

Introduction 2

An Overview of the History of Computers 2

Elements of a Computer System 4

Hardware 4

Software 6

Language of a Computer 6

Evolution of Programming Languages 8

Processing a Java Program 10

Internet, World Wide Web, Browser, and Java 13

Programming with the Problem Analysis–Coding–Execution Cycle 13

Programming Methodologies 19

Structured Programming 19

Object-Oriented Programming 19

Quick Review 21

Exercises 23

2 BASIC ELEMENTS OF JAVA 25

A Java Program 26

Basics of a Java Program 28
Comments 29
Special Symbols 30
Reserved Words (Keywords) 30
Identifiers 31

Data Types 32
Primitive Data Types 32

Arithmetic Operators and Operator Precedence 36
Order of Precedence 39

Expressions 40
Mixed Expressions 41

Type Conversion (Casting) 43

class String 45
Strings and the Operator + 46

Input 48
Allocating Memory with Named Constants and Variables 48
Putting Data into Variables 51
Declaring and Initializing Variables 55
Input (Read) Statement 56
Reading a Single Character 61

Increment and Decrement Operators 64

Output 66

Packages, Classes, Methods, and the import Statement 71

Creating a Java Application Program 72

Debugging: Understanding and Fixing Syntax Errors 77

Programming Style and Form 80
Syntax 80

Avoiding Bugs: Consistent, Proper Formatting and Code Walk-Through 84

More on Assignment Statements (Optional) 85

Quick Review 94

Exercises 97

Programming Exercises 106

3 **INTRODUCTION TO OBJECTS AND INPUT/OUTPUT** 113

Objects and Reference Variables 114

Using Predefined Classes and Methods in a Program 118

Dot Between Class (Object) Name and Class Member: A Precaution 120

`class String` 121

Input/Output 129

Formatting Output with `printf` 129

Using Dialog Boxes for Input/Output 139

Formatting the Output Using the `String` Method `format` 146

File Input/Output 149

Storing (Writing) Output to a File 152

Debugging: Understanding Logic Errors and Debugging with `print` or `println` Statements 163

Quick Review 165

Exercises 167

Programming Exercises 171

4 **CONTROL STRUCTURES I: SELECTION** 177

Control Structures 178

Relational Operators 180

Relational Operators and Primitive Data Types 181

Logical (Boolean) Operators and Logical Expressions 183

Order of Precedence 185

`boolean` Data Type and Logical (Boolean) Expressions 189

Selection: `if` and `if...else` 190
One-Way Selection 190
Two-Way Selection 193
Compound (Block of) Statements 197
Multiple Selections: Nested `if` 198
Comparing `if...else` Statements with a Series of `if` Statements 200
Short-Circuit Evaluation 201
Comparing Floating-Point Numbers for Equality: A Precaution 202
Conditional Operator (`? :`) (Optional) 204

Avoiding Bugs by Avoiding Partially Understood Concepts and Techniques 204

Program Style and Form (Revisited): Indentation 208

`switch` Structures 208

Avoiding Bugs by Avoiding Partially Understood Concepts and Techniques (Revisited) 215

Comparing Strings 223
Strings, the Assignment Operator, and the Operator `new` 229

Quick Review 230

Exercises 232

Programming Exercises 241

5 CONTROL STRUCTURES II: REPETITION 249

Why Is Repetition Needed? 250

`while` Looping (Repetition) Structure 251
Designing `while` Loops 254
Counter-Controlled `while` Loops 255
Sentinel-Controlled `while` Loops 257
Flag-Controlled `while` Loops 263
EOF-Controlled `while` Loops 266
More on Expressions in `while` Statements 271

`for` Looping (Repetition) Structure 278

`do...while` **Looping (Repetition) Structure** 288
Choosing the Right Looping Structure 293

`break` **and** `continue` **Statements** 293

Avoiding Bugs by Avoiding Patches 295

Debugging Loops 298

Nested Control Structures 299

Quick Review 304

Exercises 306

Programming Exercises 319

6 GRAPHICAL USER INTERFACE (GUI) AND OBJECT-ORIENTED DESIGN (OOD) 327

Graphical User Interface (GUI) Components 328
Creating a Window 332
`JFrame` 332
Getting Access to the Content Pane 338
`JLabel` 339
`JTextField` 343
`JButton` 347

Object-Oriented Design 363
A Simplified OOD Methodology 364

Implementing Classes and Operations 370
Primitive Data Types and the Wrapper Classes 370

Quick Review 377

Exercises 378

Programming Exercises 381

7 USER-DEFINED METHODS 383

Predefined Methods 384
Using Predefined Methods in a Program 388

User-Defined Methods 391
Value-Returning Methods 391

return Statement 395
Final Program 398

Flow of Execution 404

Void Methods 407

Primitive Data Type Variables as Parameters 411

Reference Variables as Parameters 414
Parameters and Memory Allocation 414
Reference Variables of the **String** Type as Parameters:
A Precaution 414
The **class StringBuffer** 418

Primitive Type Wrapper Classes as Parameters 421

Scope of an Identifier Within a Class 422

Method Overloading: An Introduction 427

Debugging: Using Drivers and Stubs 440

Avoiding Bugs: One-Piece-at-a-Time Coding 442

Quick Review 442

Exercises 445

Programming Exercises 456

8 USER-DEFINED CLASSES AND ADTS 465

Classes and Objects 466
Constructors 471
Unified Modeling Language Class Diagrams 472
Variable Declaration and Object Instantiation 473
Accessing Class Members 475
Built-in Operations on Classes 476
Assignment Operator and Classes: A Precaution 476
Class Scope 478

Methods and Classes 479
Definitions of the Constructors and Methods of the class Clock 479

Classes and the Method toString 494

Copy Constructor 500

Static Members of a Class 501
static Variables (Data Members) of a Class 503

Finalizers 507

Accessor and Mutator Methods 507

Debugging—Designing and Documenting a Class 510

Reference this (Optional) 512
Cascaded Method Calls (Optional) 514

Inner Classes 517

Abstract Data Types 517

Quick Review 537

Exercises 538

Programming Exercises 547

9 ARRAYS 551

Why Do We Need Arrays? 552

Arrays 553
Alternate Ways to Declare an Array 555
Accessing Array Elements 555
Specifying Array Size during Program Execution 557
Array Initialization during Declaration 558
Arrays and the Instance Variable **length** 558
Processing One-Dimensional Arrays 559
Array Index Out of Bounds Exception 564
Declaring Arrays as Formal Parameters to Methods 564
Assignment Operator, Relational Operators, and Arrays: A Precaution 565
Arrays as Parameters to Methods 567

Searching an Array for a Specific Item 572

Arrays of Objects 574

Arrays of **string** Objects 574

Arrays of Objects of Other Classes 576

Arrays and Variable Length Parameter List (Optional) 581

Two-Dimensional Arrays 589

Accessing Array Elements 591

Two-Dimensional Array Initialization during Declaration 594

Processing Two-Dimensional Arrays 595

Passing Two-Dimensional Arrays as Parameters to Methods 599

Multidimensional Arrays 603

`class Vector` **(Optional)** 616

Primitive Data Types and the **class Vector** 620

Vector Objects and the foreach Loop 620

Quick Review 621

Exercises 623

Programming Exercises 634

10 INHERITANCE AND POLYMORPHISM 639

Inheritance 640

Using Methods of the Superclass in a Subclass 642

Constructors of the Superclass and Subclass 648

Protected Members of a Class 657

Protected Access vs Package Access 660

`class Object` 661

Java Stream Classes 663

Polymorphism 664

Operator **instanceof** 670

Abstract Methods and Classes 674

Interfaces 681

Polymorphism Via Interfaces 682

Composition (Aggregation) 684

Quick Review 709

Exercises 712

Programming Exercises 719

11 **HANDLING EXCEPTIONS AND EVENTS** 723

Handling Exceptions Within a Program 724

Java's Mechanism of Exception Handling 727

`try`/`catch`/`finally` Block 728

Java Exception Hierarchy 733

Java's Exception Classes 736

Checked and Unchecked Exceptions 741

More Examples of Exception Handling 743

`class` **`Exception`** and the Operator `instanceof` 746

Rethrowing and Throwing an Exception 749

Method `printStackTrace` 753

Exception-Handling Techniques 755

Terminate the Program 755

Fix the Error and Continue 756

Log the Error and Continue 757

Creating Your Own Exception Classes 758

Event Handling 760

Quick Review 775

Exercises 777

Programming Exercises 781

12 ADVANCED GUIS AND GRAPHICS 783

Applets 787

class Font 791

class Color 794

class Graphics 800
Converting an Application Program to an Applet 808

Additional GUI Components 811
JTextArea 811
JCheckBox 816
JRadioButton 823
JComboBox 828
JList 833

Layout Managers 839
FlowLayout 840
BorderLayout 843

Menus 844

Key and Mouse Events 847
Key Events 848
Mouse Events 850

Quick Review 865

Exercises 866

Programming Exercises 868

13 RECURSION 873

Recursive Definitions 874
Direct and Indirect Recursion 876
Infinite Recursion 877
Designing Recursive Methods 877

Problem Solving Using Recursion 878
Tower of Hanoi: Analysis 887

Recursion or Iteration? 888

Quick Review 896

Exercises 897

Programming Exercises 901

14 SEARCHING AND SORTING 907

List Processing 908

- Searching 908
- Selection Sort 909
- Insertion Sort 913
- Binary Search 917

Quick Review 934

Exercises 934

Programming Exercises 936

APPENDIX A: JAVA RESERVED WORDS 939

APPENDIX B: OPERATOR PRECEDENCE 941

APPENDIX C: CHARACTER SETS 945

- ASCII (American Standard Code for Information Interchange), the First 128 Characters of the Unicode Character Set 945
- EBCDIC (Extended Binary Coded Decimal Interchange Code) 946

APPENDIX D: ADDITIONAL JAVA TOPICS 949

Binary (Base 2) Representation of a Nonnegative Integer 949

- Converting a Base 10 Number to a Binary Number (Base 2) 949
- Converting a Binary Number (Base 2) to Base 10 951
- Converting a Binary Number (Base 2) to Octol (Base 8) and Hexdecimal (Base 16) 952

Executing Java Programs Using the Command-Line Statements 954
Setting the Path in Windows 7.0 (Professional) 954
Executing Java Programs 959
Java Style Documentation 964

Creating Your Own Packages 966
Multiple File Programs 969

Formatting the Output of Decimal Numbers Using the class `DecimalFormat` 969

Packages and User-Defined Classes 972
Primitive Type Classes 972
Class: **`IntClass`** 972
Class: **`LongClass`** 976
Class: **`CharClass`** 977
Class: **`FloatClass`** 977
Class: **`DoubleClass`** 978
Class: **`BooleanClass`** 979
Using Primitive Type Classes in a Program 980

`Enumeration Types` 981

APPENDIX E: ANSWERS TO ODD-NUMBERED EXERCISES 997

Chapter 1 997

Chapter 2 998

Chapter 3 1001

Chapter 4 1002

Chapter 5 1004

Chapter 6 1007

Chapter 7 1008

Chapter 8 1010

Chapter 9 1014

Chapter 10 1016

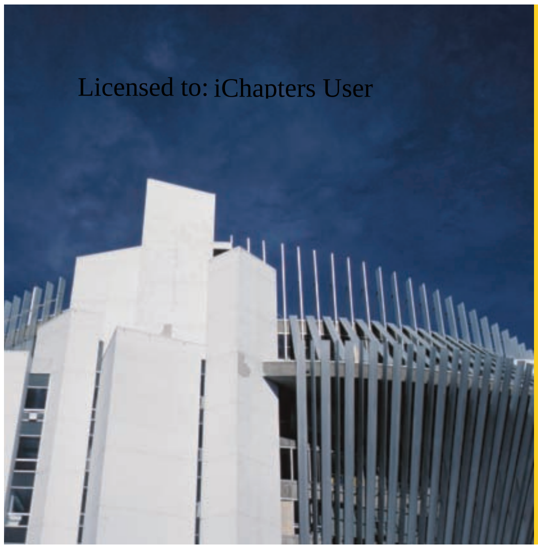
Chapter 11 1018

Chapter 12 1019

Chapter 13 1020

Chapter 14 1020

INDEX **1023**



PREFACE TO THE FIFTH EDITION

Welcome to *Java Programming: From Problem Analysis to Program Design, Fifth Edition*. Designed for a first Computer Science (CS1) Java course, this text will provide a breath of fresh air to you and your students. The CS1 course serves as the cornerstone of the Computer Science curriculum. My primary goal is to motivate and excite all programming students, regardless of their level. Motivation breeds excitement for learning. Motivation and excitement are critical factors that lead to the success of the programming student. This text is the culmination and development of my classroom notes throughout more than fifty semesters of teaching successful programming.

Warning: This text can be expected to create a serious reduction in the demand for programming help during your office hours. Other side effects include significantly diminished student dependency on others while learning to program.

The primary focus in writing this text is on student learning. Therefore, in addition to clear explanations, we address the key issues that otherwise impede student learning. For example, a common question that arises naturally during an early programming assignment is: "How many variables and what kinds are needed in this program?" We illustrate this important and crucial step by helping students learn why variables are needed and how data in a variable is manipulated. Next students learn that the analysis of the problem will spill the number and types of the variables. Once students grasp this key concept, control structures (selection and loops) become easier to learn. The second major impediment in learning programming is parameter passing. We pay special attention to this topic. First students learn how to use predefined methods and how actual and formal parameters relate. Next students learn about user-defined methods. They see visual diagrams that help them learn how methods are called and how formal parameters affect actual parameters. Once students have a clear understanding of these two key concepts, they readily assimilate advanced topics.

The topics are introduced at a pace that is conducive to learning. The writing style is friendly, engaging, and straightforward. It parallels the learning style of the contemporary CS1 student. Before introducing a key concept, the student learns why the concept is needed, and then sees examples illustrating the concept. Special attention is paid to topics that are essential in mastering the Java programming language and in acquiring a foundation for further study of computer science.

Other important topics include debugging techniques and techniques for avoiding programming bugs. When a beginner compiles his/her first program and sees that the number of errors exceeds the length of this first program, he/she becomes frustrated by the plethora of errors, only some of which can be interpreted. To ease this frustration and help students learn to produce correct programs, debugging and bug avoidance techniques are presented systematically throughout the text.

Changes in the Fifth Edition

The main changes are:

- In the fifth edition, new debugging sections have been added and some of the old ones have been rewritten. These sections are indicated with a debugging icon.
- The fifth edition contains more than 125 new exercises, 27 new programming exercises, and numerous new examples spread throughout the book.
- In Chapters 6 and 12 the GUI figures have been captured and replaced in Windows 7 Professional environment.
- Appendix D contains screen images illustrating how to compile and execute a Java program using the command-line statements as well as how to set the path in Windows 7 Professional environment.

These changes were implemented based on comments from the text reviewers of the fifth edition. The source code and the programming exercises are developed and tested using Java 6.0 and the version of Java 7.0 available at the time this book was being typeset.

Approach

Once conceived as a Web programming language, Java slowly but surely found its way into classrooms where it now serves as a first programming language in computer science curricula (CS1). Java is a combination of traditional style programming—programming with a non-graphical user interface—and modern style programming with a graphical user interface (GUI). This book introduces you to both styles of programming. After giving a brief description of each chapter, we discuss how to read this book.

Chapter 1 briefly reviews the history of computers and programming languages. The reader can quickly skim and become familiar with some of the hardware and software components of the computer. This chapter also gives an example of a Java program and describes how a Java program is processed. The two basic problem-solving techniques, structured programming and object-oriented design, are also presented.

After completing Chapter 2, students become familiar with the basics of Java and are ready to write programs that are complicated enough to do some computations. The debugging section in this chapter illustrates how to interpret and correct syntax errors.

The three terms that you will encounter throughout the book are—primitive type variables, reference variables, and objects. Chapter 3 makes clear distinctions between these terms and sets the tone for the rest of the book. An object is a fundamental entity in an object-oriented programming language. This chapter further explains how an object works. The **class** `String` is one of the most important classes in Java. This chapter introduces this class and explains how various methods of this class can be used to manipulate strings. Because input/output is fundamental to any programming language, it is introduced early, and is covered in detail in Chapter 3. The debugging section in this chapter illustrates how to find and correct logical errors.

Chapters 4 and 5 introduce control structures used to alter the sequential flow of execution. The debugging sections in these chapters discuss and illustrate logical errors associated with selection and looping structures.

Java is equipped with powerful yet easy-to-use graphical user interface (GUI) components to create user-friendly graphical programs. Chapter 6 introduces various GUI components and gives examples of how to use these components in Java application programs. Because Java is an object-oriented programming language, the second part of Chapter 6 discusses and gives examples of how to solve various problems using object-oriented design methodology.

Chapter 7 discusses user-defined methods. Parameter passing is a fundamental concept in any programming language. Several examples, including visual diagrams, help readers understand this concept. It is recommended that readers with no prior programming background spend extra time on this concept. The debugging section in this chapter discuss how to debug a program using stubs and drivers.

Chapter 8 discusses user-defined classes. In Java, a class is an important and widely used element. It is used to create Java programs, group related operations, and it allows users to create their own data types. This chapter uses extensive visual diagrams to illustrate how objects of classes manipulate data.

Chapter 9 describes arrays. This chapter also introduces variable length formal parameter lists. In addition, this chapter introduces foreach loops and explains how this loop can be used to process the elements of an array. This chapter also discusses the sequential searching algorithm and the **class** `Vector`.

Inheritance is an important principle of object-oriented design. It encourages code reuse. Chapter 10 discusses inheritance and gives various examples to illustrate how classes are derived from existing classes. In addition, this chapter also discusses polymorphism, abstract classes, inner classes, and composition.

An occurrence of an undesirable situation that can be detected during program execution is called an exception. For example, division by zero is an exception. Java provides extensive support for handing exceptions. Chapter 11 shows how to handle exceptions in a program. Chapter 11 also discusses event handling, which was introduced in Chapter 6. Chapter 12 picks up the discussion of GUI components started in Chapter 6. This chapter introduces additional GUI components and discusses how to create applets.

Chapter 13 introduces recursion. Several examples illustrate how recursive methods execute.

Chapter 14 discusses a binary search algorithm as well as bubble sort, selection sort, insertion sort, and quick sort algorithms. Additional content covering the sorting algorithms bubble sort and quick sort is provided online at *www.cengagebrain.com*.

Appendix A lists the reserved words in Java. Appendix B shows the precedence and associativity of the Java operators. Appendix C lists the ASCII (American Standard Code for Information Interchange) portion of the Unicode character set as well as the EBCDIC (Extended Binary Code Decimal Interchange) character set.

Appendix D contains additional topics in Java. The topics covered are converting a base 10 number to binary (base 2) number and vice versa, converting a number from base 2 to base 8 (base 16) and vice versa, how to compile and execute a Java program using command line statements, how to create Java style documentation of the user-defined classes, how to create packages, how to use user-defined classes in a Java program, and `enum` type. Appendix E gives answers to the odd-numbered exercises in the text. Those odd-numbered exercises with very long solutions will not be in the text, but will be provided to students online at *www.cengagebrain.com*.

How To Use This Book

Java is a complex and very powerful language. In addition to traditional (non-GUI) programming, Java provides extensive support for creating programs that use a graphical user interface (GUI). Chapter 3 introduces graphical input and output dialog boxes. Chapter 6 introduces the most commonly used GUI components such as labels, buttons, and text fields. More extensive coverage of GUI components is provided in Chapter 12.

This book can be used in two ways. One way is an integrated approach in which readers learn how to write both non-GUI and GUI programs as they learn basic programming concepts and skills. The other approach focuses on illustrating fundamental programming concepts with non-GUI programming first, and later incorporating GUI components. The recommended chapter sequence for each of these approaches is as follows:

- **Integrated approach:** Study all chapters in sequence.
- **Non-GUI first, then GUI:** Study Chapters 1–5 in sequence. Then study Chapters 7–11 and Chapters 13 and 14. This approach initially skips Chapters 6 and 12, the primary GUI chapters. After studying Chapters 1–5, 7–11, 13, and 14, the reader can come back to study Chapters 6 and 12, the GUI chapters. Also note that Chapter 14 can be studied after Chapter 9.

If you choose the second approach, it should also be noted that the Programming Examples in Chapters 8 and 10 are developed first without any GUI components, and then the programs are extended to incorporate GUI components. Also, if Chapter 6 is skipped, the reader can skip the event handling part of Chapter 11. Chapter 13 (recursion) contains two Programming Examples: one creates a non-GUI application program, while the other creates a program that uses GUI. If you skip Chapters 6 and 12, you can skip the GUI part of the Programming Examples in Chapters 8, 10, 11, and 13. Once you have studied Chapter 6 and 12, you can study the GUI part of the Programming Examples of Chapters 8, 10, 11, and 13.

Figure 1 shows a chapter dependency diagram for this book. Solid arrows indicate that the chapter at the beginning of the arrow is required before studying the chapter at the end of the arrow. A dotted arrow indicates that the chapter at the beginning of the arrow is not essential to studying the chapter at the end of the dotted arrow.

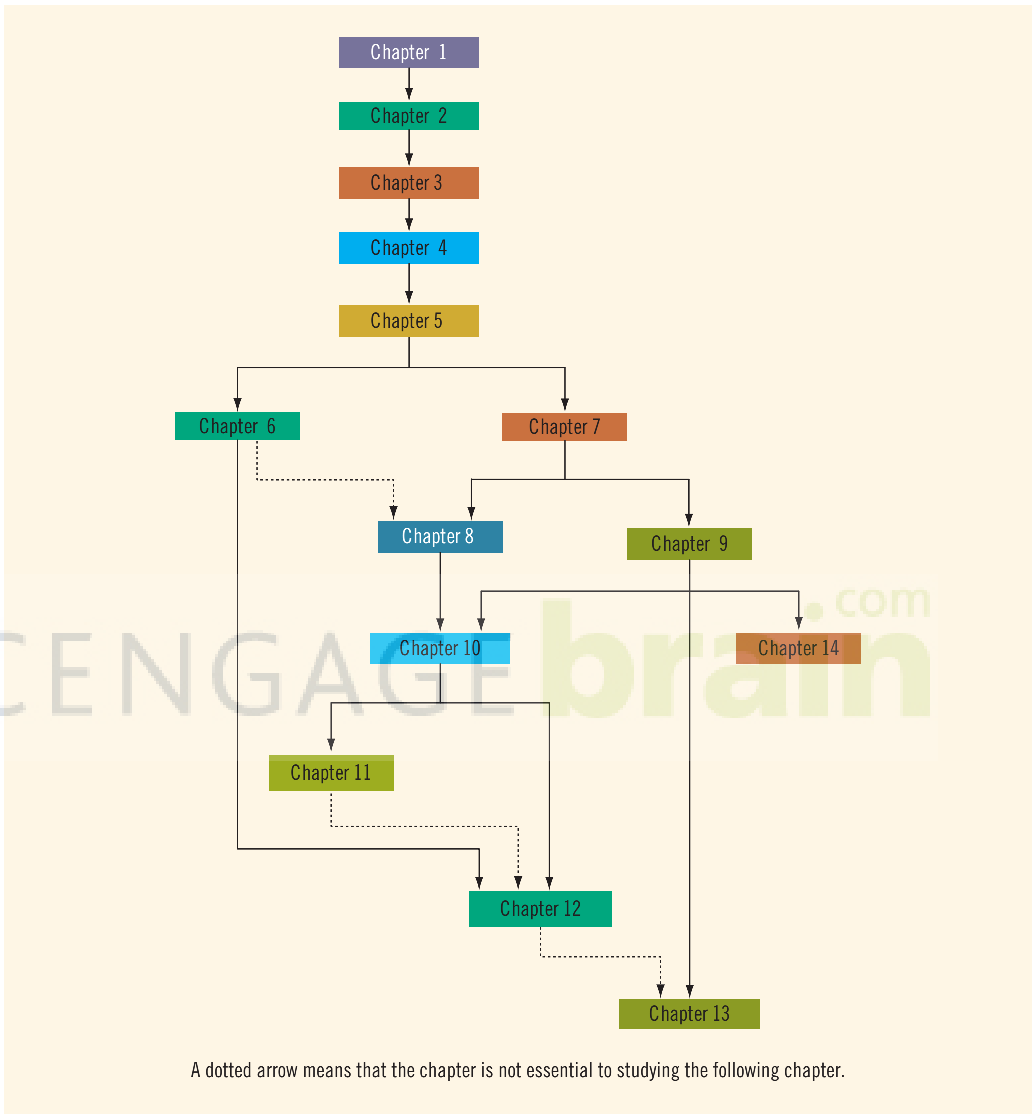
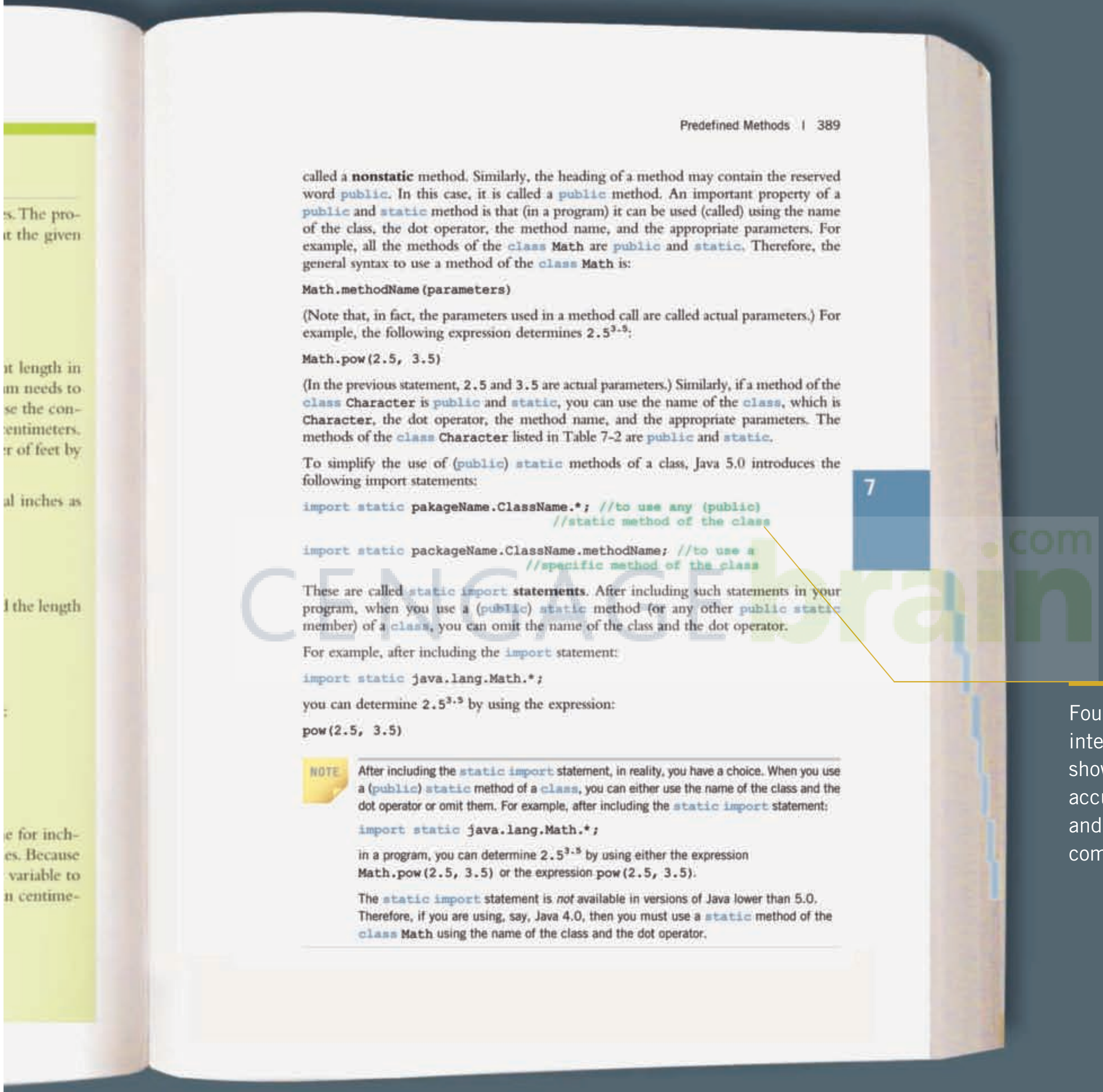


FIGURE 1 Chapter dependency diagram

All source code and solutions have been written, compiled, and quality assurance tested with Java 6.0 and the version of Java 7.0 available at the time this book was being typeset.

Features of the Book



called a **nonstatic** method. Similarly, the heading of a method may contain the reserved word `public`. In this case, it is called a `public` method. An important property of a `public` and `static` method is that (in a program) it can be used (called) using the name of the class, the dot operator, the method name, and the appropriate parameters. For example, all the methods of the `class Math` are `public` and `static`. Therefore, the general syntax to use a method of the `class Math` is:

```
Math.methodName(parameters)
```

(Note that, in fact, the parameters used in a method call are called actual parameters.) For example, the following expression determines $2.5^{3.5}$:

```
Math.pow(2.5, 3.5)
```

(In the previous statement, `2.5` and `3.5` are actual parameters.) Similarly, if a method of the `class Character` is `public` and `static`, you can use the name of the `class`, which is `Character`, the dot operator, the method name, and the appropriate parameters. The methods of the `class Character` listed in Table 7-2 are `public` and `static`.

To simplify the use of (`public`) `static` methods of a class, Java 5.0 introduces the following import statements:

```
import static pakageName.ClassName.*; //to use any (public)
                                      //static method of the class

import static packageName.ClassName.methodName; //to use a
                                                //specific method of the class
```

These are called `static import` **statements**. After including such statements in your program, when you use a (`public`) `static` method (or any other `public static` member) of a `class`, you can omit the name of the class and the dot operator.

For example, after including the `import` statement:

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

you can determine $2.5^{3.5}$ by using the expression:

```
pow(2.5, 3.5)
```

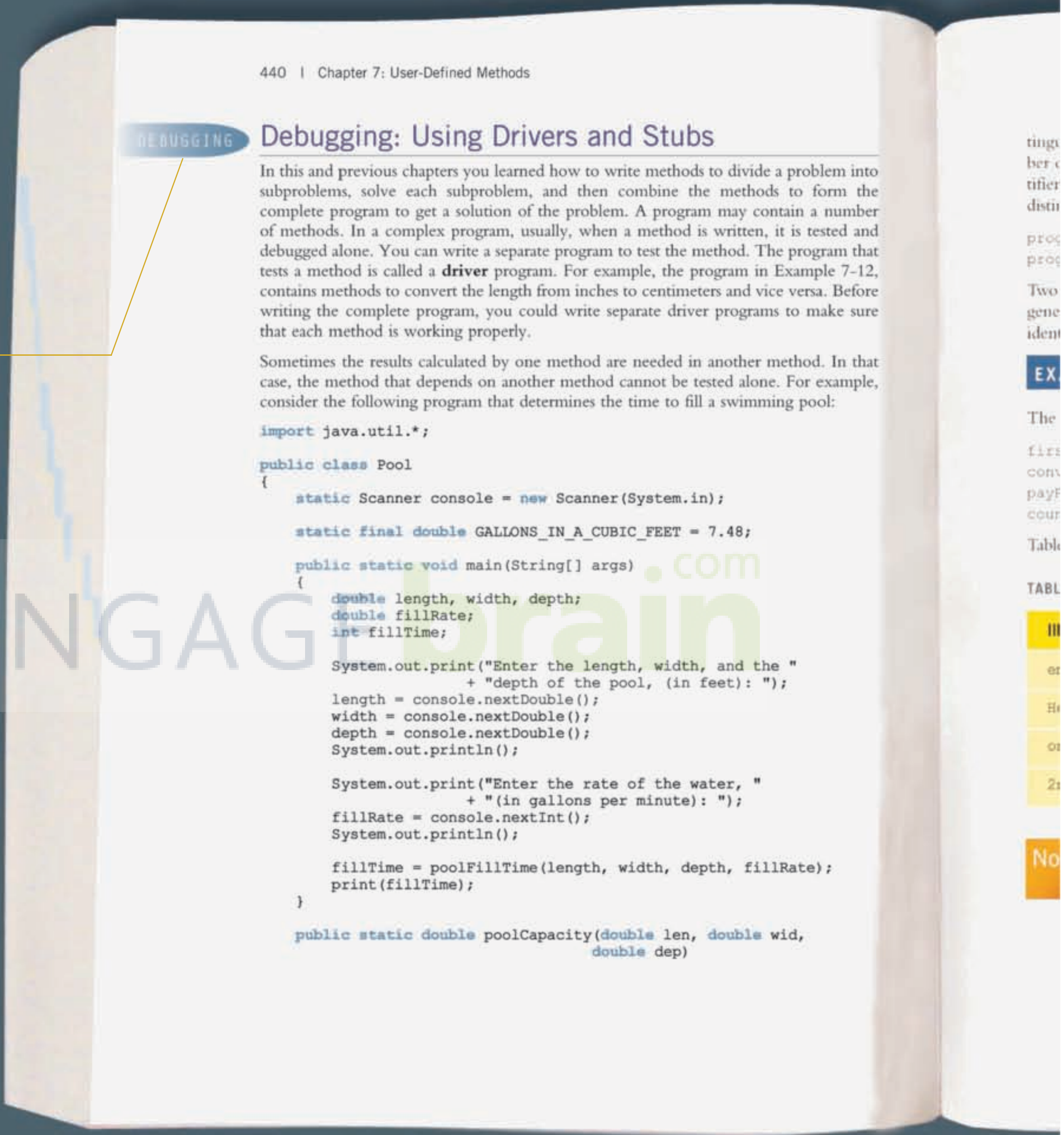
NOTE After including the `static import` statement, in reality, you have a choice. When you use a (`public`) `static` method of a `class`, you can either use the name of the class and the dot operator or omit them. For example, after including the `static import` statement:

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

in a program, you can determine $2.5^{3.5}$ by using either the expression `Math.pow(2.5, 3.5)` or the expression `pow(2.5, 3.5)`.

The `static import` statement is *not* available in versions of Java lower than 5.0. Therefore, if you are using, say, Java 4.0, then you must use a `static` method of the `class Math` using the name of the class and the dot operator.

Four-color interior design shows accurate code and related comments.



DEBUGGING

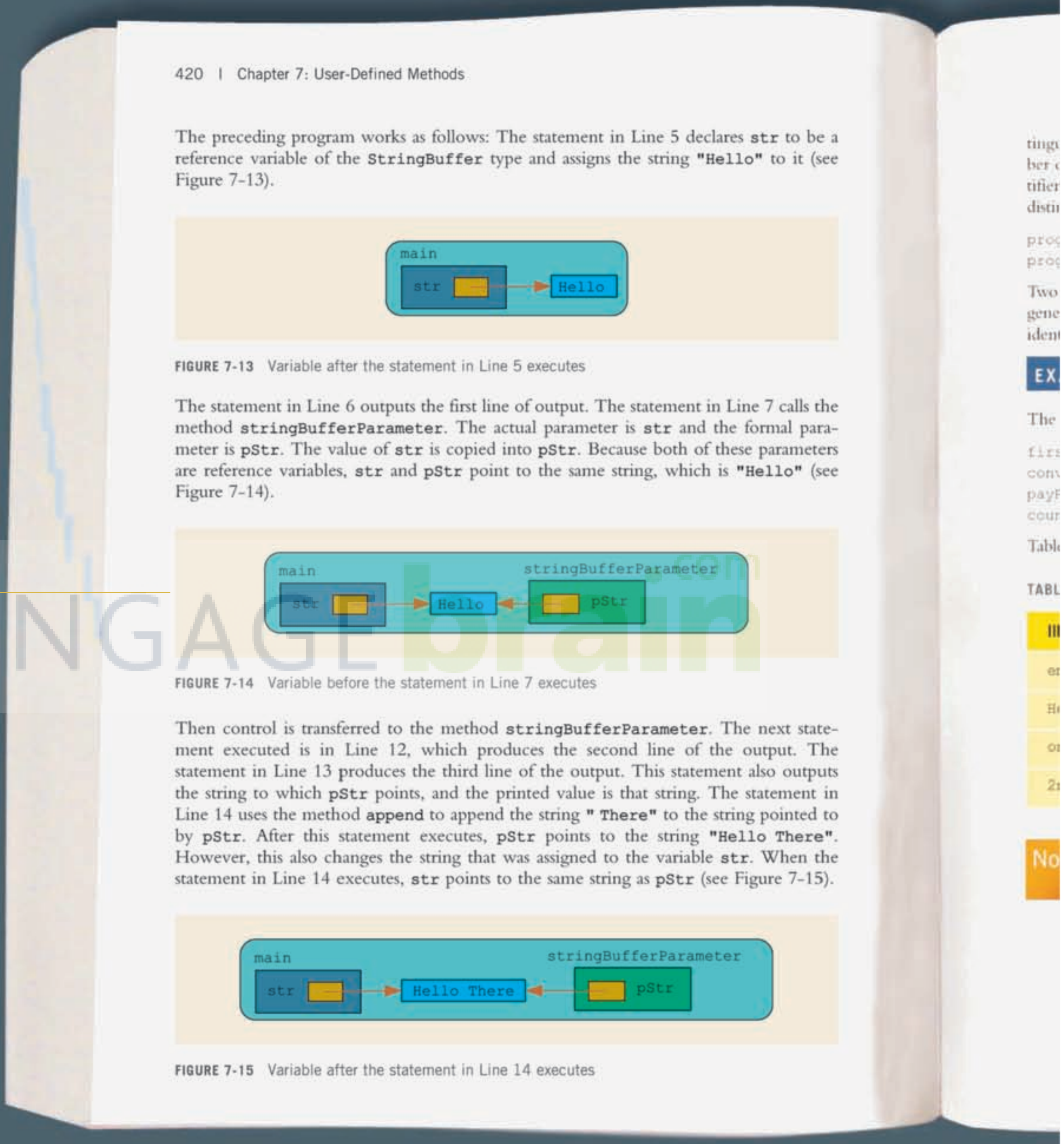
Debugging: Using Drivers and Stubs

In this and previous chapters you learned how to write methods to divide a problem into subproblems, solve each subproblem, and then combine the methods to form the complete program to get a solution of the problem. A program may contain a number of methods. In a complex program, usually, when a method is written, it is tested and debugged alone. You can write a separate program to test the method. The program that tests a method is called a **driver** program. For example, the program in Example 7-12, contains methods to convert the length from inches to centimeters and vice versa. Before writing the complete program, you could write separate driver programs to make sure that each method is working properly.

Sometimes the results calculated by one method are needed in another method. In that case, the method that depends on another method cannot be tested alone. For example, consider the following program that determines the time to fill a swimming pool:

```
import java.util.*;

public class Pool
{
    static Scanner console = new Scanner(System.in);

    static final double GALLONS_IN_A_CUBIC_FEET = 7.48;

    public static void main(String[] args)
    {
        double length, width, depth;
        double fillRate;
        int fillTime;

        System.out.print("Enter the length, width, and the "
                       + "depth of the pool, (in feet): ");
        length = console.nextDouble();
        width = console.nextDouble();
        depth = console.nextDouble();
        System.out.println();

        System.out.print("Enter the rate of the water, "
                       + "(in gallons per minute): ");
        fillRate = console.nextInt();
        System.out.println();

        fillTime = poolFillTime(length, width, depth, fillRate);
        print(fillTime);
    }

    public static double poolCapacity(double len, double wid,
                                      double dep)
```

The debugging sections show how to find and correct syntax and semantic (logical) errors.



420 | Chapter 7: User-Defined Methods

The preceding program works as follows: The statement in Line 5 declares **str** to be a reference variable of the **StringBuffer** type and assigns the string **"Hello"** to it (see Figure 7-13).

FIGURE 7-13 Variable after the statement in Line 5 executes

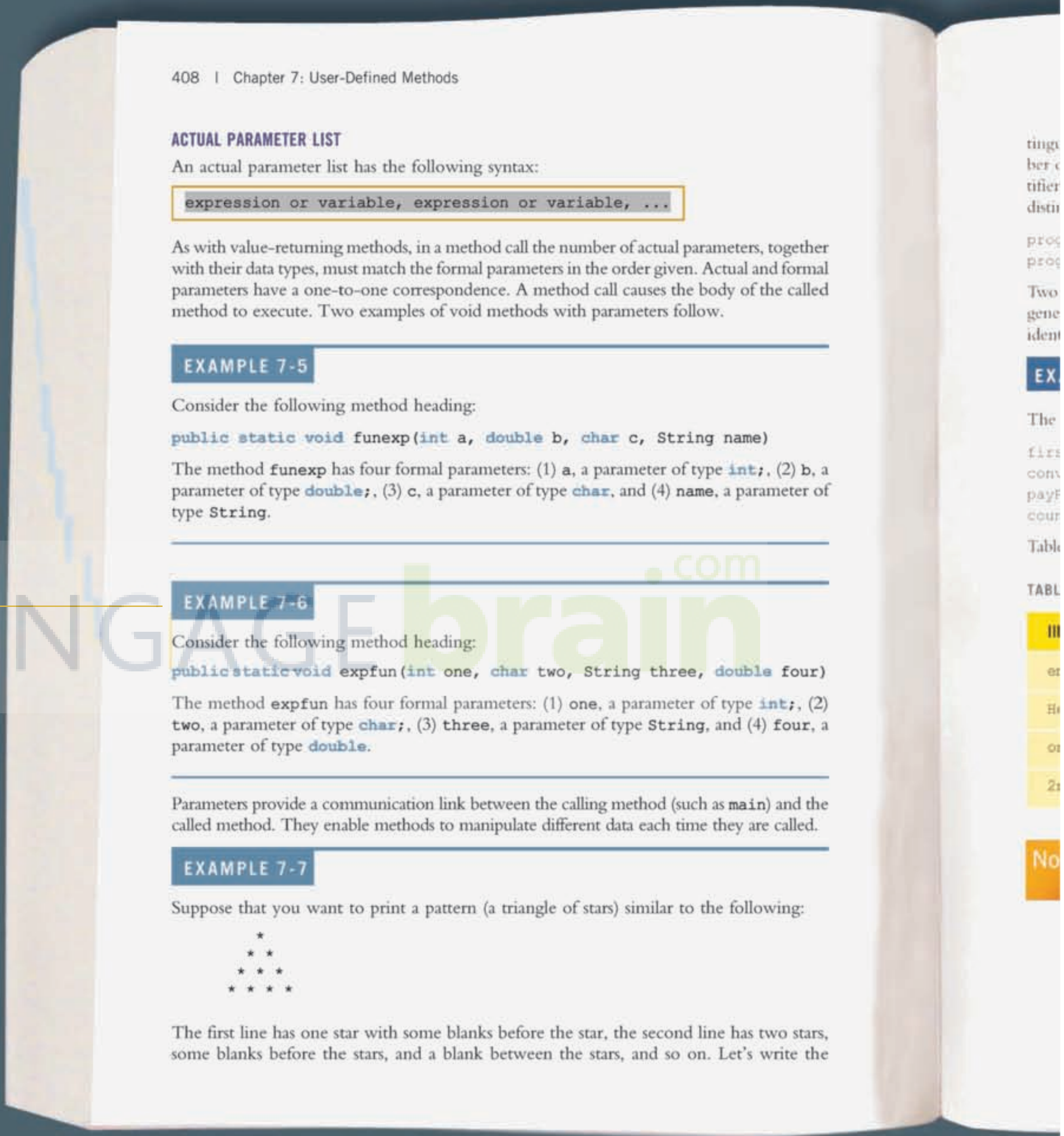
The statement in Line 6 outputs the first line of output. The statement in Line 7 calls the method **stringBufferParameter**. The actual parameter is **str** and the formal parameter is **pStr**. The value of **str** is copied into **pStr**. Because both of these parameters are reference variables, **str** and **pStr** point to the same string, which is **"Hello"** (see Figure 7-14).

FIGURE 7-14 Variable before the statement in Line 7 executes

Then control is transferred to the method **stringBufferParameter**. The next statement executed is in Line 12, which produces the second line of the output. The statement in Line 13 produces the third line of the output. This statement also outputs the string to which **pStr** points, and the printed value is that string. The statement in Line 14 uses the method **append** to append the string **" There"** to the string pointed to by **pStr**. After this statement executes, **pStr** points to the string **"Hello There"**. However, this also changes the string that was assigned to the variable **str**. When the statement in Line 14 executes, **str** points to the same string as **pStr** (see Figure 7-15).

FIGURE 7-15 Variable after the statement in Line 14 executes

More than 250 visual diagrams, both extensive and exhaustive, illustrate difficult concepts.



ACTUAL PARAMETER LIST

An actual parameter list has the following syntax:

```
expression or variable, expression or variable, ...
```

As with value-returning methods, in a method call the number of actual parameters, together with their data types, must match the formal parameters in the order given. Actual and formal parameters have a one-to-one correspondence. A method call causes the body of the called method to execute. Two examples of void methods with parameters follow.

EXAMPLE 7-5

Consider the following method heading:

```
public static void funexp(int a, double b, char c, String name)
```

The method **funexp** has four formal parameters: (1) **a**, a parameter of type **int**;, (2) **b**, a parameter of type **double**;, (3) **c**, a parameter of type **char**, and (4) **name**, a parameter of type **String**.

EXAMPLE 7-6

Consider the following method heading:

```
public static void expfun(int one, char two, String three, double four)
```

The method **expfun** has four formal parameters: (1) **one**, a parameter of type **int**;, (2) **two**, a parameter of type **char**;, (3) **three**, a parameter of type **String**, and (4) **four**, a parameter of type **double**.

Parameters provide a communication link between the calling method (such as **main**) and the called method. They enable methods to manipulate different data each time they are called.

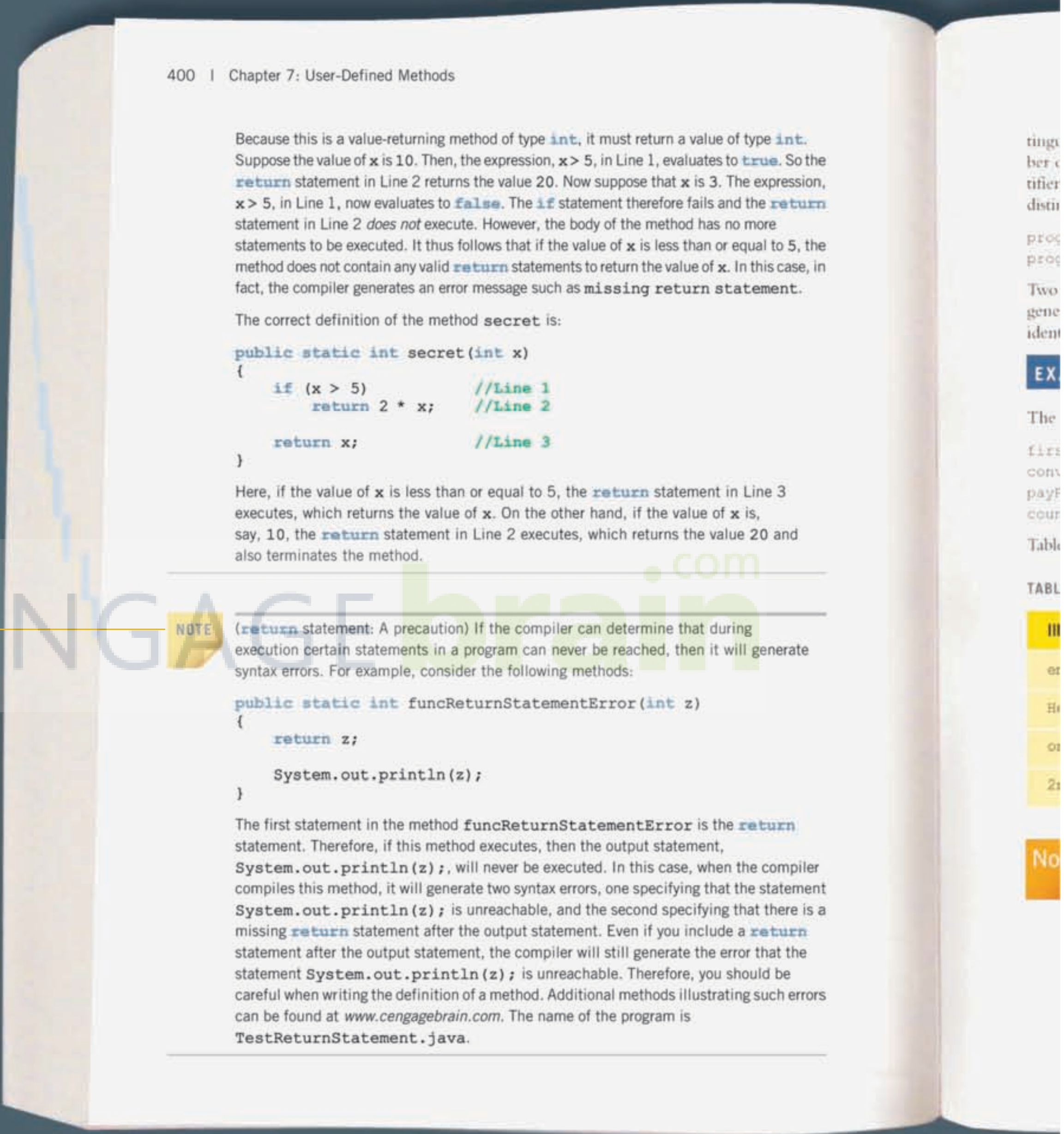
EXAMPLE 7-7

Suppose that you want to print a pattern (a triangle of stars) similar to the following:

```
   *
  * *
 * * *
* * * *
```

The first line has one star with some blanks before the star, the second line has two stars, some blanks before the stars, and a blank between the stars, and so on. Let's write the

Numbered Examples illustrate the key concepts with their relevant code. The programming code in these examples is followed by a Sample Run. An explanation then follows that describes what each line in the code does.



400 | Chapter 7: User-Defined Methods

Because this is a value-returning method of type `int`, it must return a value of type `int`. Suppose the value of **x** is 10. Then, the expression, **x** > 5, in Line 1, evaluates to `true`. So the `return` statement in Line 2 returns the value 20. Now suppose that **x** is 3. The expression, **x** > 5, in Line 1, now evaluates to `false`. The `if` statement therefore fails and the `return` statement in Line 2 *does not* execute. However, the body of the method has no more statements to be executed. It thus follows that if the value of **x** is less than or equal to 5, the method does not contain any valid `return` statements to return the value of **x**. In this case, in fact, the compiler generates an error message such as `missing return statement`.

The correct definition of the method `secret` is:

```
public static int secret(int x)
{
    if (x > 5)              //Line 1
        return 2 * x;       //Line 2

    return x;               //Line 3
}
```

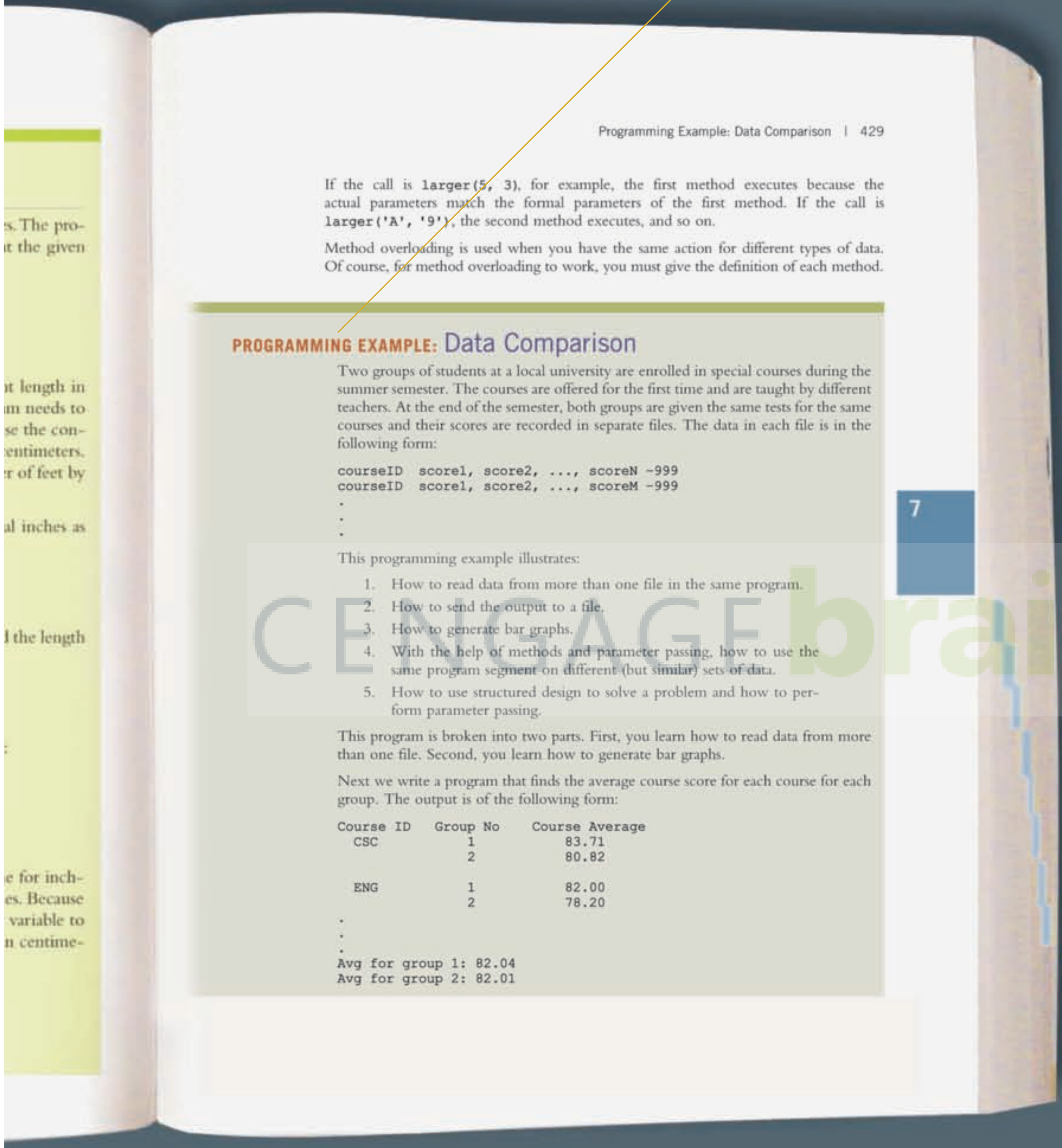
Here, if the value of **x** is less than or equal to 5, the `return` statement in Line 3 executes, which returns the value of **x**. On the other hand, if the value of **x** is, say, 10, the `return` statement in Line 2 executes, which returns the value 20 and also terminates the method.

NOTE (`return` statement: A precaution) If the compiler can determine that during execution certain statements in a program can never be reached, then it will generate syntax errors. For example, consider the following methods:

```
public static int funcReturnStatementError(int z)
{
    return z;

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

The first statement in the method `funcReturnStatementError` is the `return` statement. Therefore, if this method executes, then the output statement, `System.out.println(z);`, will never be executed. In this case, when the compiler compiles this method, it will generate two syntax errors, one specifying that the statement `System.out.println(z);` is unreachable, and the second specifying that there is a missing `return` statement after the output statement. Even if you include a `return` statement after the output statement, the compiler will still generate the error that the statement `System.out.println(z);` is unreachable. Therefore, you should be careful when writing the definition of a method. Additional methods illustrating such errors can be found at *www.cengagebrain.com*. The name of the program is `TestReturnStatement.java`.

Notes highlight important facts about the concepts introduced in the chapter.



If the call is **larger(5, 3)**, for example, the first method executes because the actual parameters match the formal parameters of the first method. If the call is **larger('A', '9')**, the second method executes, and so on.

Method overloading is used when you have the same action for different types of data. Of course, for method overloading to work, you must give the definition of each method.

PROGRAMMING EXAMPLE: Data Comparison

Two groups of students at a local university are enrolled in special courses during the summer semester. The courses are offered for the first time and are taught by different teachers. At the end of the semester, both groups are given the same tests for the same courses and their scores are recorded in separate files. The data in each file is in the following form:

```
courseID  score1, score2, ..., scoreN -999
courseID  score1, score2, ..., scoreM -999
.
.
.
```

This programming example illustrates:

1. How to read data from more than one file in the same program.
2. How to send the output to a file.
3. How to generate bar graphs.
4. With the help of methods and parameter passing, how to use the same program segment on different (but similar) sets of data.
5. How to use structured design to solve a problem and how to perform parameter passing.

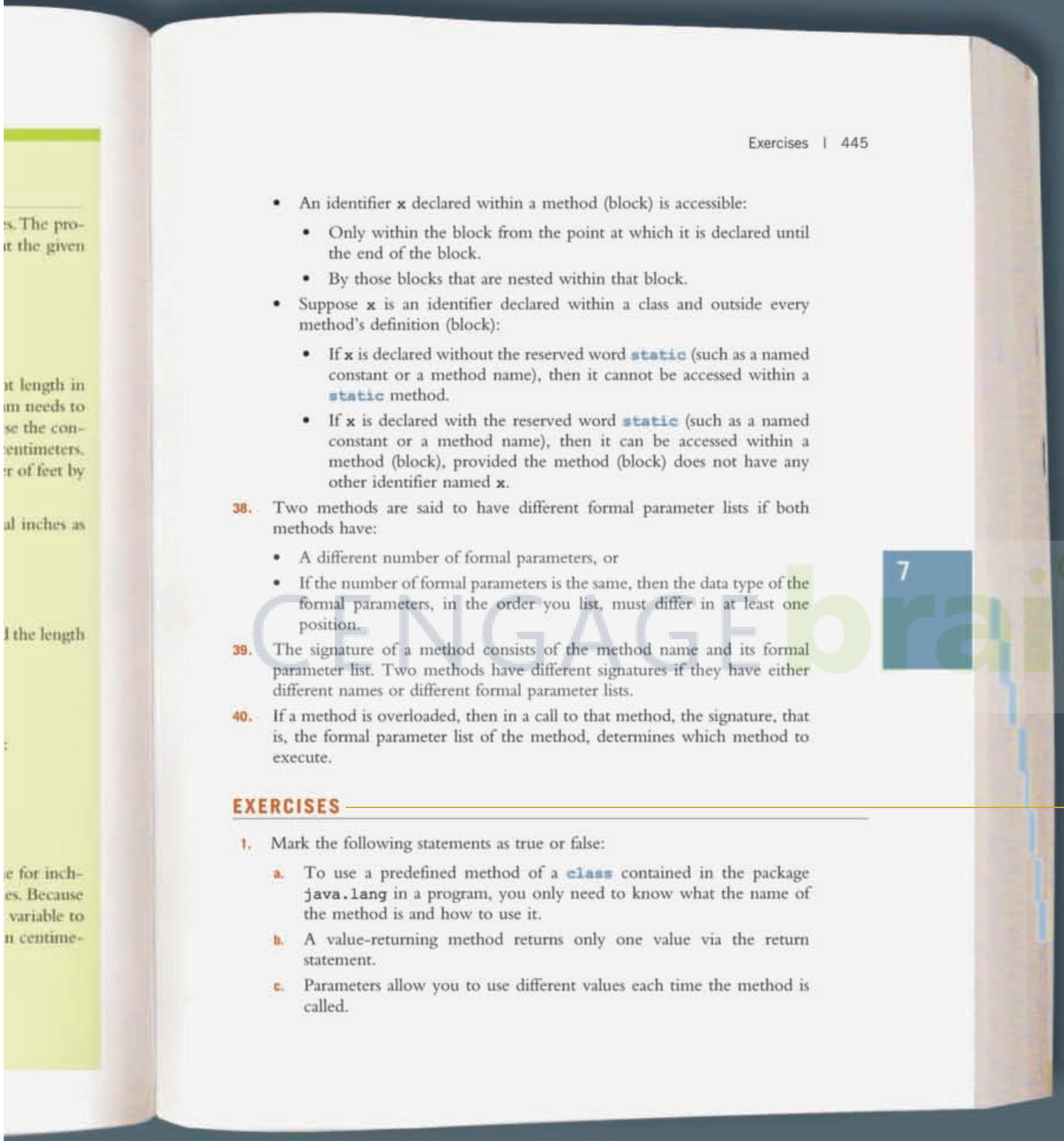
This program is broken into two parts. First, you learn how to read data from more than one file. Second, you learn how to generate bar graphs.

Next we write a program that finds the average course score for each course for each group. The output is of the following form:

```
Course ID   Group No    Course Average
  CSC          1           83.71
               2           80.82

  ENG          1           82.00
               2           78.20
.
.
.
Avg for group 1: 82.04
Avg for group 2: 82.01
```

7

Programming Examples are complete programs featured in each chapter. These examples include the accurate, concrete stages of Input, Output, Problem Analysis and Algorithm Design, and a Complete Program Listing.



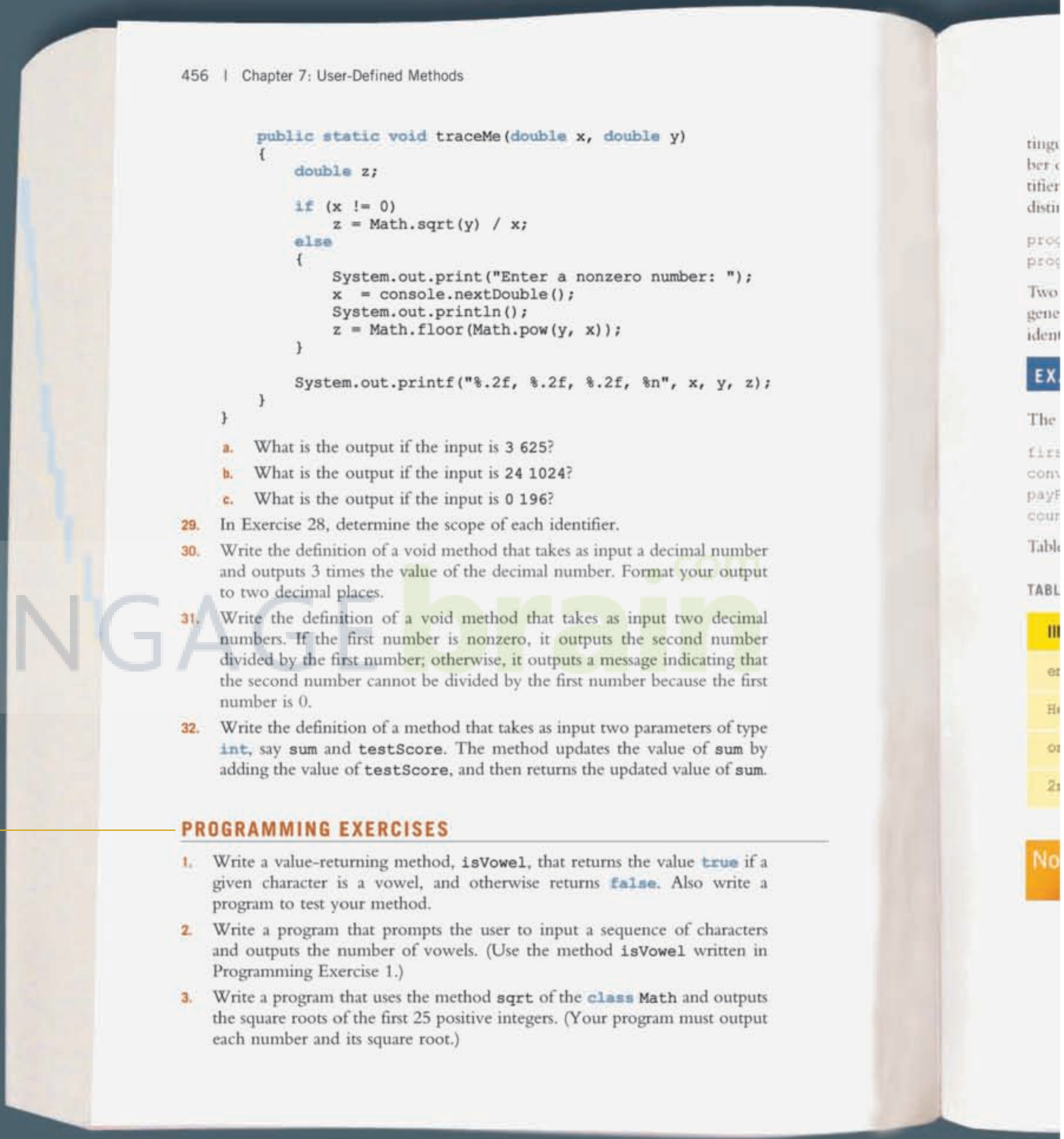
- An identifier **x** declared within a method (block) is accessible:
  - Only within the block from the point at which it is declared until the end of the block.
  - By those blocks that are nested within that block.
- Suppose **x** is an identifier declared within a class and outside every method's definition (block):
  - If **x** is declared without the reserved word **static** (such as a named constant or a method name), then it cannot be accessed within a **static** method.
  - If **x** is declared with the reserved word **static** (such as a named constant or a method name), then it can be accessed within a method (block), provided the method (block) does not have any other identifier named **x**.

38. Two methods are said to have different formal parameter lists if both methods have:
    - A different number of formal parameters, or
    - If the number of formal parameters is the same, then the data type of the formal parameters, in the order you list, must differ in at least one position.
39. The signature of a method consists of the method name and its formal parameter list. Two methods have different signatures if they have either different names or different formal parameter lists.
40. If a method is overloaded, then in a call to that method, the signature, that is, the formal parameter list of the method, determines which method to execute.

EXERCISES

1. Mark the following statements as true or false:
   a. To use a predefined method of a `class` contained in the package `java.lang` in a program, you only need to know what the name of the method is and how to use it.
   b. A value-returning method returns only one value via the return statement.
   c. Parameters allow you to use different values each time the method is called.

Exercises further reinforce learning and ensure that students have, in fact, mastered the material.



```
    public static void traceMe(double x, double y)
    {
        double z;

        if (x != 0)
            z = Math.sqrt(y) / x;
        else
        {
            System.out.print("Enter a nonzero number: ");
            x  = console.nextDouble();
            System.out.println();
            z = Math.floor(Math.pow(y, x));
        }

        System.out.printf("%.2f, %.2f, %.2f, %n", x, y, z);
    }
}
```

a. What is the output if the input is `3 625`?

b. What is the output if the input is `24 1024`?

c. What is the output if the input is `0 196`?

29. In Exercise 28, determine the scope of each identifier.

30. Write the definition of a void method that takes as input a decimal number and outputs 3 times the value of the decimal number. Format your output to two decimal places.

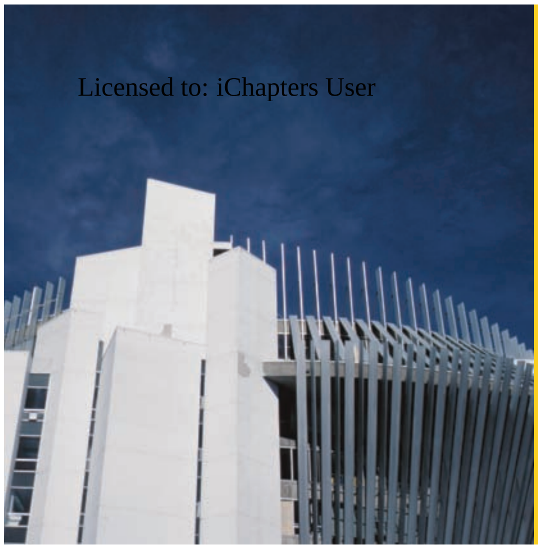
31. Write the definition of a void method that takes as input two decimal numbers. If the first number is nonzero, it outputs the second number divided by the first number; otherwise, it outputs a message indicating that the second number cannot be divided by the first number because the first number is 0.

32. Write the definition of a method that takes as input two parameters of type `int`, say `sum` and `testScore`. The method updates the value of `sum` by adding the value of `testScore`, and then returns the updated value of `sum`.

PROGRAMMING EXERCISES

1. Write a value-returning method, `isVowel`, that returns the value `true` if a given character is a vowel, and otherwise returns `false`. Also write a program to test your method.
2. Write a program that prompts the user to input a sequence of characters and outputs the number of vowels. (Use the method `isVowel` written in Programming Exercise 1.)
3. Write a program that uses the method `sqrt` of the `class Math` and outputs the square roots of the first 25 positive integers. (Your program must output each number and its square root.)

Programming Exercises challenge students to write Java programs with a specified outcome.



Supplemental Resources

The following supplemental materials are available when this book is used in a classroom setting.

Most instructor teaching tools, outlined below, are available with this book on a single CD-ROM, and are also available for instructor access at *login.cengage.com*.

Electronic Instructor's Manual

The Instructor's Manual that accompanies this textbook includes:

- Additional instructional material to assist in class preparation, including suggestions for lecture topics.
- Solutions to all the end-of-chapter materials, including the Programming Exercises.

ExamView®

This textbook is accompanied by ExamView, a powerful testing software package that allows instructors to create and administer printed, computer (LAN-based), and Internet exams. ExamView includes hundreds of questions that correspond to the topics covered in this text, enabling students to generate detailed study guides that include page references for further review. These computer-based and Internet testing components allow students to take exams at their computers, and save the instructor time because each exam is graded automatically.

PowerPoint Presentations

Microsoft PowerPoint slides are available for each chapter. These slides are provided as a teaching aid for classroom presentations, either to make available to students on the network for chapter review, or to be printed for classroom distribution. Instructors can add their own slides for additional topics that they introduce to the class.

Distance Learning

Course Technology is proud to present online courses in WebCT and Blackboard to provide the most complete and dynamic learning experience possible. For more information on how to bring distance learning to your course, contact your local Course Technology sales representative.

Source Code

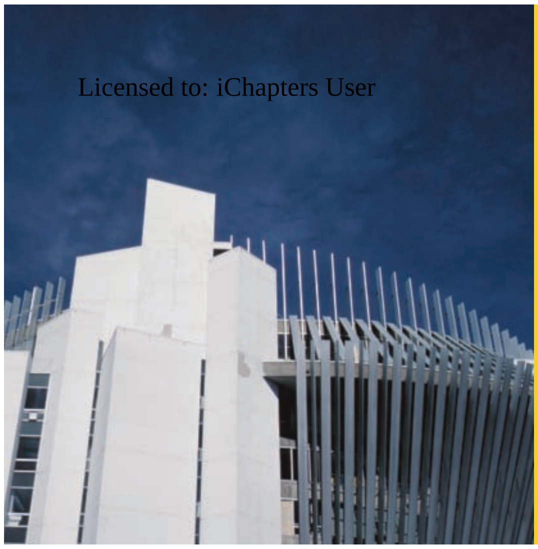
The source code is available for students at *www.cengagebrain.com*. At the *cengagebrain.com* home page, search for the ISBN of your title (from the back cover of your book) using the search box at the top of the page. This will take you to the product page where these resources can be found. The source code is also available on the Instructor Resources CD-ROM. The input files needed to run some of the programs are also included with the source code.

Additional Student Files

The Additional Student Files referenced throughout the text are available on the Instructor Resources CD. Students can download these files directly at *www.cengagebrain.com*. At the *cengagebrain.com* home page, search for the ISBN of your title using the search box at the top of the page. This will take you to the product page where these resources can be found. Click the *Access Now* link below the book cover to find all study tools and additional files available directly to students. Additional Student Files appear on the left navigation and provide access to additional Java programs, selected solutions, and more.

Solution Files

The solution files for all programming exercises are available for instructor download at *http://login.cengage.com* and are also available on the Instructor Resources CD-ROM. The input files needed to run some of the programming exercises are also included with the solution files.



ACKNOWLEDGMENTS

There are many people I must thank who, in one way or another, contributed to the success of this book. First, I would like to thank those who e-mailed numerous comments that helped to improve on the fourth edition. I am thankful to Professors S.C. Cheng and Randall Crist for constantly supporting this project.

I owe a great deal to the following reviewers, who patiently read each page of every chapter of the current version and made critical comments that helped to improve the book: Nadimpalli Mahadev, Fitchburg State College and Baoqiang Yan, Missouri Western State University. Additionally, I would like to thank Brian Candido, Springfield Technical Community College, for his review of the proposal package. The reviewers will recognize that their suggestions have not been overlooked and, in fact, made this a better book.

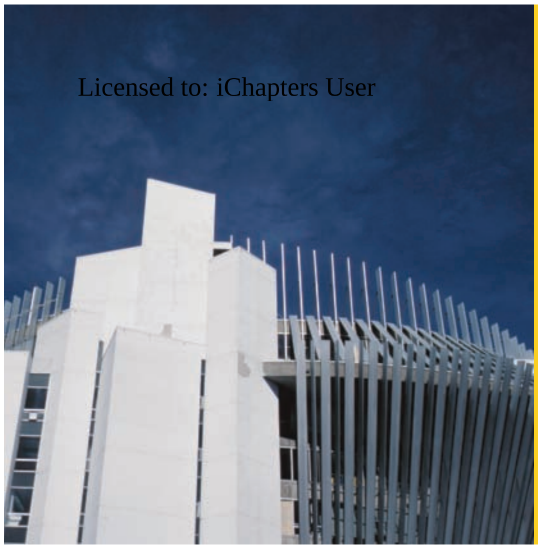
Next, I express thanks to Brandi Shailer, Acquisitions Editor, for recognizing the importance and uniqueness of this project. All this would not have been possible without the careful planning of Senior Product Manager Alyssa Pratt. I extend my sincere thanks to Alyssa, as well as to Content Project Manager, Lisa Weidenfeld. I also thank Sreejith Govindan of Integra Software Services for assisting us in keeping the project on schedule. I would like to thank Chris Scriver and Serge Palladino of the MQA department of Course Technology for patiently and carefully proofreading the text, testing the code, and discovering typos and errors.

I am thankful to my parents for their blessings.

Finally, I am thankful to the support of my wife Sadhana, and especially my daughter Shelly, to whom this book is dedicated. They cheered me up whenever I was overwhelmed during the writing of this book.

We welcome any comments concerning the text. Comments may be forwarded to the following e-mail address: `malik@creighton.edu`.

D.S. Malik

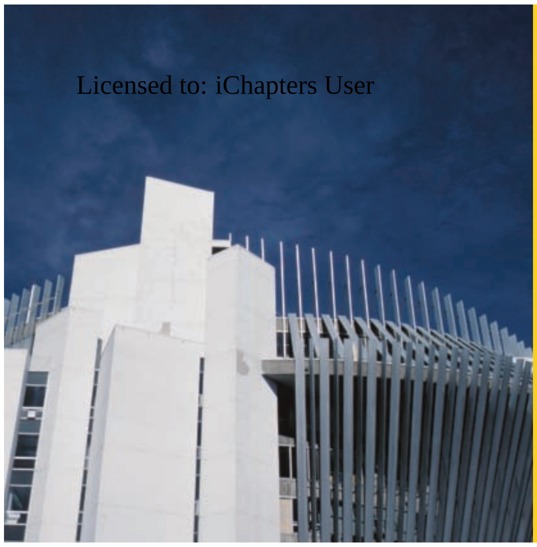


APPENDIX A JAVA RESERVED WORDS

The following table lists Java reserved words in alphabetical order.

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

The reserved words `const` and `goto` are *not* currently in use.



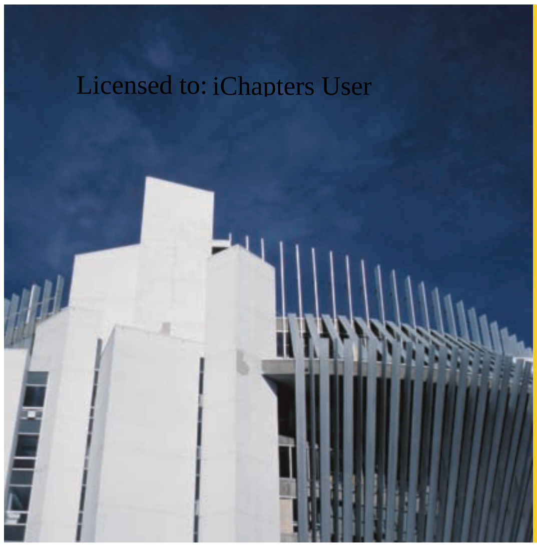
APPENDIX B OPERATOR PRECEDENCE

The following table shows the precedence of operators in Java from highest to lowest, and their associativity.

Operator	Description	Precedence Level	Associativity
.	Object member access	1	Left to right
[ ]	Array subscripting	1	Left to right
(parameters)	Method call	1	Left to right
++	Postincrement	1	Left to right
--	Postdecrement	1	Left to right
++	Preincrement	2	Right to left
--	Predecrement	2	Right to left
+	Unary plus	2	Right to left
-	Unary minus	2	Right to left
!	Logical not	2	Right to left
~	Bitwise not	2	Right to left
new	Object instantiation	3	Right to left
(type)	Type conversion	3	Right to left
*	Multiplication	4	Left to right
/	Division	4	Left to right
%	Remainder (modulus)	4	Left to right

Operator	Description	Precedence Level	Associativity
+	Addition	5	Left to right
–	Subtraction	5	Left to right
+	String concatenation	5	Left to right
<<	Left shift	6	Left to right
>>	Right shift with sign extension	6	Left to right
>>>	Right shift with zero extension	6	Left to right
<	Less than	7	Left to right
<=	Less than or equal to	7	Left to right
>	Greater than	7	Left to right
>=	Greater than or equal to	7	Left to right
`instanceof`	Type comparison	7	Left to right
==	Equal to	8	Left to right
!=	Not equal to	8	Left to right
&	Bitwise AND	9	Left to right
&	Logical AND	9	Left to right
^	Bitwise XOR	10	Left to right
^	Logical XOR	10	Left to right
|	Bitwise OR	11	Left to right
|	Logical OR	11	Left to right
&&	Logical AND	12	Left to right

Operator	Description	Precedence Level	Associativity
`||`	Logical OR	13	Left to right
`?:`	Conditional operator	14	Right to left
`=`	Assignment	15	Right to left
	Compound Operators		
`+=`	Addition, then assignment	15	Right to left
`+=`	String concatenation, then assignment	15	Right to left
`-=`	Subtraction, then assignment	15	Right to left
`*=`	Multiplication, then assignment	15	Right to left
`/=`	Division, then assignment	15	Right to left
`%=`	Remainder, then assignment	15	Right to left
`<<=`	Bitwise left shift, then assignment	15	Right to left
`>>=`	Bitwise right shift, then assignment	15	Right to left
`>>>=`	Bitwise unsigned-right shift, then assignment	15	Right to left
`&=`	Bitwise AND, then assignment	15	Right to left
`&=`	Logical AND, then assignment	15	Right to left
`|=`	Bitwise OR, then assignment	15	Right to left
`|=`	Logical OR, then assignment	15	Right to left
`^=`	Bitwise XOR, then assignment	15	Right to left
`^=`	Logical XOR, then assignment	15	Right to left



APPENDIX C CHARACTER SETS

This appendix lists and describes the character sets for ASCII (American Standard Code for Information Interchange), which also comprises the first 128 characters of the Unicode character set, and EBCDIC (Extended Binary Coded Decimal Interchange Code).

ASCII (American Standard Code for Information Interchange), the First 128 Characters of the Unicode Character Set

The following table shows the first 128 characters of the Unicode (ASCII) character set.

ASCII

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	lf	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	b	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

NOTE For more information on the Unicode/ASCII character set, visit the Web site at *http://www.unicode.org*.

Note that the character b at position 32 represents the space character. The first 32 characters, that is, the characters at positions 00–31 and at position 127 are nonprintable characters. The following table shows the abbreviations and meanings of these characters.

nul	null character	ff	form feed	can	cancel
soh	start of header	cr	carriage return	em	end of medium
stx	start of text	so	shift out	sub	substitute
etx	end of text	si	shift in	esc	escape
eot	end of transmission	dle	data link escape	fs	file separator
enq	enquiry	dc1	device control 1	gs	group separator
ack	acknowledge	dc2	device control 2	rs	record separator
bel	bell	dc3	device control 3	us	unit separator
bs	backspace	dc4	device control 4	b	space
ht	horizontal tab	nak	negative acknowledge	del	delete
lf	line feed	syn	synchronous idle		
vt	vertical tab	etb	end of transmitted block		

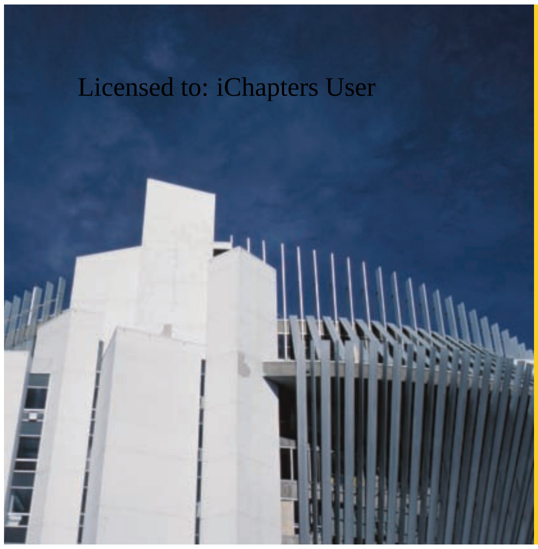
EBCDIC (Extended Binary Coded Decimal Interchange Code)

The following table shows some of the characters in the EBCDIC character set.

EBCDIC										
	0	1	2	3	4	5	6	7	8	9
6					b					
7						.	<	(	+	|
8	&									
9	!	$	*	)	;	¬	-	/		
10								'	%	_

EBCDIC										
11	>	?								
12		`	:	#	@	'	=	"		a
	0	1	2	3	4	5	6	7	8	9
13	b	c	d	e	f	g	h	i		
14						j	k	l	m	n
15	o	p	q	r						
16		~	s	t	u	v	w	x	y	z
17										
18	[	]								
19				A	B	C	D	E	F	G
20	H	I								J
21	K	L	M	N	O	P	Q	R		
22							S	T	U	V
23	W	X	Y	Z						
24	0	1	2	3	4	5	6	7	8	9

The numbers 6–24 in the first column specify the left digit(s) and the numbers 0–9 in the second row specify the right digits of the characters in the EBCDIC data set. For example, the character in the row marked 19 (the number in the first column) and the column marked 3 (the number in the second row) is `A`. Therefore, the character at position 193 (which is the 194th character) is `A`. Moreover, the character b at position 64 represents the space character. This table does not show all the characters in the EBCDIC character set. In fact, the characters at positions 00–63 and 250–255 are nonprintable control characters.



APPENDIX D
ADDITIONAL JAVA TOPICS

Binary (Base 2) Representation of a Nonnegative Integer

Converting a Base 10 Number to a Binary Number (Base 2)

Chapter 1 noted that `A` is the 66th character in the ASCII character set, but its position is 65 because the position of the first character is `0`. Furthermore, the binary number `1000001` is the binary representation of `65`. The number system that we use daily is called the **decimal number system** or **base 10 system**. The number system that the computer uses is called the **binary number system** or **base 2 system**. In this section, we describe how to find the binary representation of a nonnegative integer and vice versa.

Consider 65. Note that:

$$65 = 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0.$$

Similarly:

$$\begin{aligned}711 = {} & 1 \times 2^9 + 0 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 \\ & + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0.\end{aligned}$$

In general, if m is a nonnegative integer, then m can be written as:

$$m = a_k \times 2^k + a_{k-1} \times 2^{k-1} + a_{k-2} \times 2^{k-2} + \cdots + a_1 \times 2^1 + a_0 \times 2^0,$$

for some nonnegative integer k, and where $a_i = 0$ or 1, for each $i = 0, 1, 2, \ldots, k$. The binary number $a_k a_{k-1} a_{k-2} \ldots a_1 a_0$ is called the **binary** or **base 2 representation** of m. In this case, we usually write:

$$m_{10} = (a_k a_{k-1} a_{k-2} \cdots a_1 a_0)_2$$

and say that m to the base 10 is $a_k a_{k-1} a_{k-2} \ldots a_1 a_0$ to the base 2.

For example, for the integer 65, $k = 6$, $a_6 = 1$, $a_5 = 0$, $a_4 = 0$, $a_3 = 0$, $a_2 = 0$, $a_1 = 0$, and $a_0 = 1$. Thus, $a_6a_5a_4a_3a_2a_1a_0 = 1000001$, so the binary representation of 65 is 1000001, that is:

$$65_{10} = (1000001)_2.$$

If no confusion arises, then we write $(1000001)_2$ as 1000001_2.

Similarly, for the number 711, $k = 9$, $a_9 = 1$, $a_8 = 0$, $a_7 = 1$, $a_6 = 1$, $a_5 = 0$, $a_4 = 0$, $a_3 = 0$, $a_2 = 1$, $a_1 = 1$, and $a_0 = 1$. Thus:

$$711_{10} = 1011000111_2.$$

It follows that to find the binary representation of a nonnegative integer, we need to find the coefficients, which are 0 or 1, of various powers of 2. However, there is an easy algorithm, described next, that can be used to find the binary representation of a nonnegative integer. First, note that:

$$0_{10} = 0_2,\ 1_{10} = 1_2,\ 2_{10} = 10_2,\ 3_{10} = 11_2,\ 4_{10} = 100_2,\ 5_{10} = 101_2,\ 6_{10} = 110_2, \text{ and } 7_{10} = 111_2.$$

Let us consider the integer 65. Note that 65 / 2 = 32 and 65 % 2 = 1, where % is the mod operator. Next, 32 / 2 = 16, and 32 % 2 = 0, and so on. It can be shown that a_0 = 65 % 2 = 1, a_1 = 32 % 2 = 0, and so on. We can show this continuous division and obtain the remainder with the help of Figure D-1.

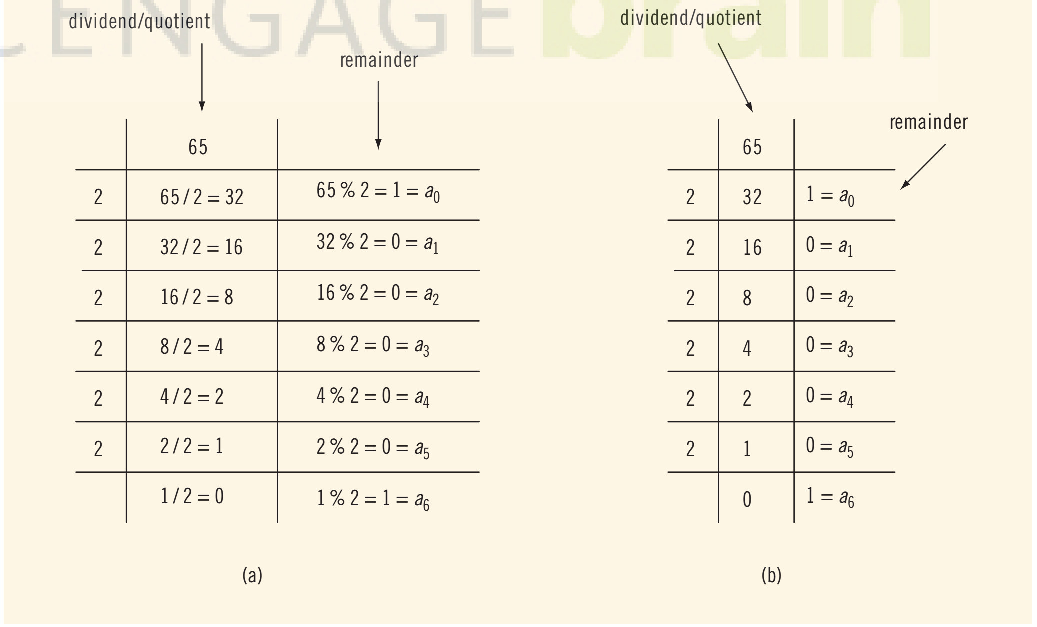
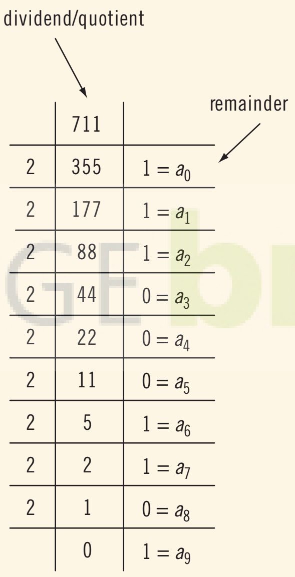

dividend/quotient
remainder
65
2 | 65 / 2 = 32 | 65 % 2 = 1 = a_0
2 | 32 / 2 = 16 | 32 % 2 = 0 = a_1
2 | 16 / 2 = 8 | 16 % 2 = 0 = a_2
2 | 8 / 2 = 4 | 8 % 2 = 0 = a_3
2 | 4 / 2 = 2 | 4 % 2 = 0 = a_4
2 | 2 / 2 = 1 | 2 % 2 = 0 = a_5
1 / 2 = 0 | 1 % 2 = 1 = a_6
(a)
dividend/quotient
remainder
65
2 | 32 | 1 = a_0
2 | 16 | 0 = a_1
2 | 8 | 0 = a_2
2 | 4 | 0 = a_3
2 | 2 | 0 = a_4
2 | 1 | 0 = a_5
0 | 1 = a_6
(b)

FIGURE D-1 Determining the binary representation of 65

Notice that in Figure D-1(a), starting at the second row, the second column contains the quotient when the number in the previous row is divided by 2, and the third column contains the remainder of that division. For example, in the second row, 65 / 2 = 32, and 65 % 2 = 1. In the third row, 32 / 2 = 16 and 32 % 2 = 0, and so on. For each row, the number in the second column is divided by 2, the quotient is written in the row below the current row, and the remainder appears in the third column. When using a figure such as D-1 to find the binary representation of a nonnegative integer, we typically show only the quotients and remainders, as shown in Figure D-1(b). You can write the binary representation of the number, starting with the last remainder in the third column, followed by the second to the last remainder, and so on. Thus:

$65_{10} = 1000001_2$.

Next, consider the number 711. Figure D-2 shows the quotients and the remainders.



	711	
2	355	$1 = a_0$
2	177	$1 = a_1$
2	88	$1 = a_2$
2	44	$0 = a_3$
2	22	$0 = a_4$
2	11	$0 = a_5$
2	5	$1 = a_6$
2	2	$1 = a_7$
2	1	$0 = a_8$
	0	$1 = a_9$

FIGURE D-2 Determining the binary representation of 711

From Figure D-2, it follows that:

$711_{10} = 1011000111_2$.

Converting a Binary Number (Base 2) to Base 10

To convert a number from base 2 to base 10, we first find the weight of each bit in the binary number, which is assigned from right to left. The weight of the rightmost bit is 0.

The weight of the bit immediately to the left of the rightmost bit is 1, the weight of the bit immediately to the left of it is 2, and so on. Consider the binary number 1001101. The weight of each bit is as follows:

Weight	6	5	4	3	2	1	0
	1	0	0	1	1	0	1

We use the weight of each bit to find the equivalent decimal number. For each bit, we multiply the bit by 2 to the power of its weight and then we add all of the numbers. For the above binary number, the equivalent decimal number is:

$$1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$= 64 + 0 + 0 + 8 + 4 + 0 + 1$$

$$= 77.$$

Converting a Binary Number (Base 2) to Octal (Base 8) and Hexadecimal (Base 16)

The previous sections described how to convert a binary number to a decimal number (base 2). Even though the language of a computer is binary, if the binary number is too long, then it will be hard to manipulate it manually. To effectively deal with binary numbers, two more number systems, octal (base 8) and hexadecimal (base 16), are of interest to computer scientists.

The digits in the octal number system are 0, 1, 2, 3, 4, 5, 6, and 7. The digits in the hexadecimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. So A in hexadecimal is 10 in decimal, B in hexadecimal is 11 in decimal, and so on.

The algorithm to convert a binary number into an equivalent number in octal (or hexadecimal) is quite simple. Before we describe the method to do so, let us review some notations. Suppose a_b represents the number a to the base b. For example, $2A0_{16}$ means 2A0 to the base 16, and 63_8 means 63 to the base 8.

First, we describe how to convert a binary number into an equivalent octal number and vice versa. Table D-1 describes the first 8 octal numbers.

TABLE D-1 Binary Representation of First 8 Octal Numbers

Binary	Octal		Binary	Octal
000	0		100	4
001	1		101	5
010	2		110	6
011	3		111	7

Consider the binary number 1101100010101. To find the equivalent octal number, starting from right to left, we consider three digits at a time and write their octal representation. Note that the binary number 1101100010101 has only 13 digits. So when we consider three digits at a time, at the end, we will be left with only one digit. In this case, we just add two 0s to the left of the binary number; the equivalent binary number is 001101100010101. Thus:

$$\begin{aligned} 1101100010101_2 &= 001101100010101_2 \\ &= 001\ 101\ 100\ 010\ 101 \\ &= 15425_8 \text{ because } 001_2 = 1_8,\ 101_2 = 5_8,\ 100_2 = 4_8,\ 010_2 = 2_8, \text{ and } 101_2 = 5_8. \end{aligned}$$

Thus, $1101100010101_2 = 15425_8$.

To convert an octal number into an equivalent binary number, using Table D-1, write the binary representation of each octal digit in the number. For example:

$$\begin{aligned} 3761_8 &= 011\ 111\ 110\ 001_2 \\ &= 011111110001_2 \\ &= 11111110001_2. \end{aligned}$$

Thus, $3761_8 = 11111110001_2$.

Next, we discuss how to convert a binary number into an equivalent hexadecimal number and vice versa. The method to do so is similar to converting a number from binary to octal and vice versa, except that here we work with four binary digits. Table D-2 gives the binary representation of the first 16 hexadecimal numbers.

TABLE D-2 Binary Representation of First 16 Hexadecimal Numbers

Binary	Hexadecimal		Binary	Hexadecimal
0000	0		1000	8
0001	1		1001	9
0010	2		1010	A
0011	3		1011	B
0100	4		1100	C
0101	5		1101	D
0110	6		1110	E
0111	7		1111	F

Consider the binary number 1111101010001010101_2. Now:

$$\begin{aligned} 1111101010001010101_2 &= 111\ 1101\ 0100\ 0101\ 0101_2 \\ &= 0111\ 1101\ 0100\ 0101\ 0101_2, \text{ add one zero to the left} \\ &= 7D455_{16}. \end{aligned}$$

Hence, $1111101010001010101_2 = 7D455_{16}$.

Next, to convert a hexadecimal number into an equivalent binary number, write the four-digit binary representation of each hexadecimal digit into that number. For example:

$$\begin{aligned} A7F32_{16} &= 1010\ 0111\ 1111\ 0011\ 0010_2 \\ &= 10100111111100110010_2. \end{aligned}$$

Thus, $A7F32_{16} = 10100111111100110010_2$.

Executing Java Programs Using the Command-Line Statements

When you install JDK 7.0 in the Windows 7.0 environment, the system creates two main subdirectories: `Java\jdk1.7.0` and `Java\jre1.7.0`. These two subdirectories are, typically, created within the directory `c:\Program Files\Java`. However, these subdirectories might also be created in the directory c: as `c:\jdk1.7.0` and `c:\jre1.7.0`. (Check your systems documentation.) The files necessary to compile and execute Java programs are placed within these subdirectories, along with other files. For example, the file `javac.exe` to compile a Java program and the file `java.exe` to execute a Java application program are placed within the subdirectory `jdk1.7.0\bin` or `jdk1.7.0\fastdebug\bin`. You can set (or alter) the Windows system environment variable `Path` to add the path where the files `javac.exe` and `java.exe` are located. This will allow you to conveniently compile a Java program from within any subdirectory. In the Windows 7.0 Professional environment, you can also set the environment variable `CLASSPATH` so that when you execute a Java program, the system can find the compiled code of the program. Next, we describe how to set up the `Path`.

Setting the Path in Windows 7.0 (Professional)

To set the `Path` so that you can compile a Java program from within any subdirectory, perform the following steps.

1. Click the `Start` button (lower-left corner of the window.)
2. Select `Control Panel`. A window similar to the window shown in Figure D-3 appears.

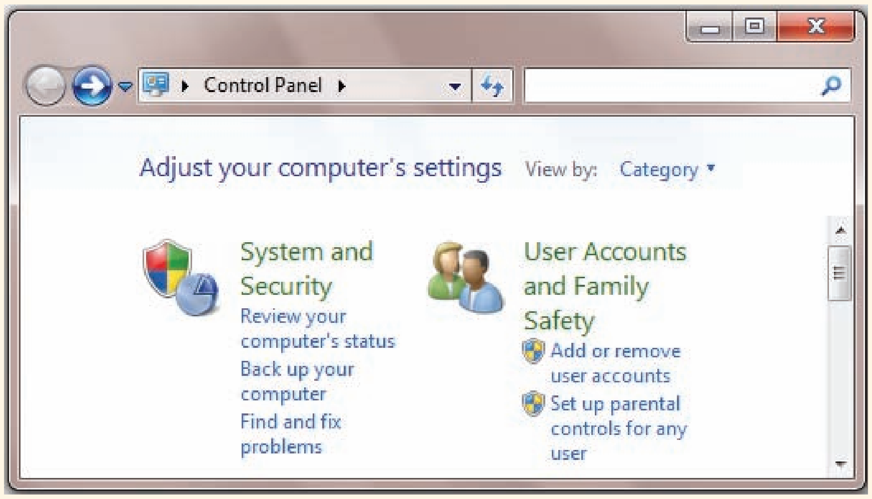

Control Panel
Adjust your computer's settings
View by: Category
System and Security
Review your computer's status
Back up your computer
Find and fix problems
User Accounts and Family Safety
Add or remove user accounts
Set up parental controls for any user

FIGURE D-3 Control Panel

3. Select the `System and Security` option. A window similar to the window shown in Figure D-4 appears.

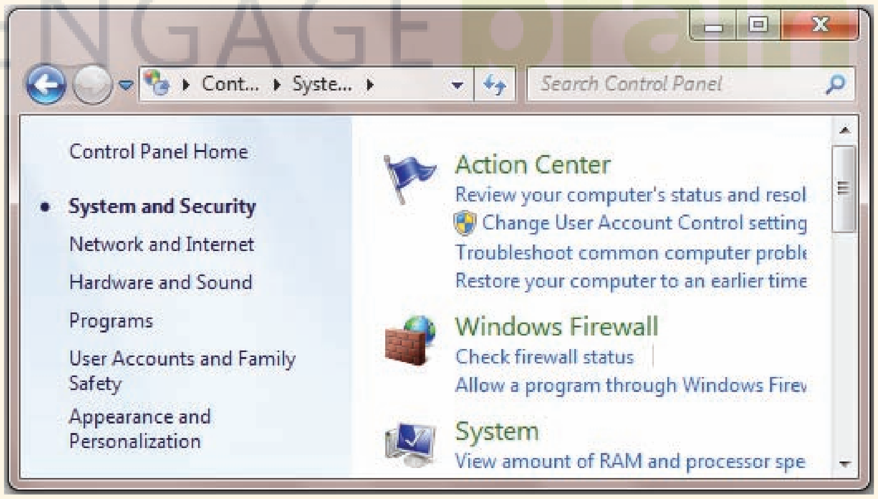
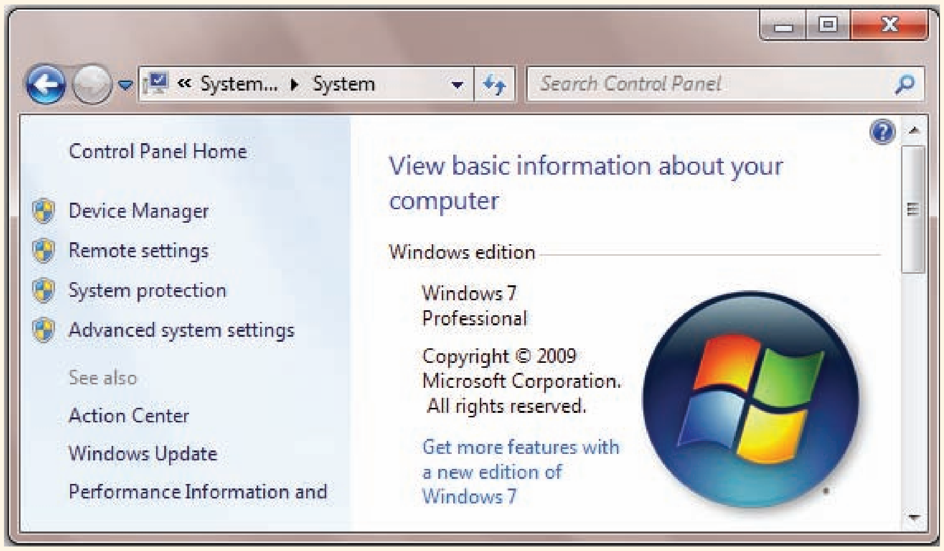

Cont... Syste...
Search Control Panel
Control Panel Home
System and Security
Network and Internet
Hardware and Sound
Programs
User Accounts and Family Safety
Appearance and Personalization
Action Center
Review your computer's status and resol
Change User Account Control setting
Troubleshoot common computer probl
Restore your computer to an earlier time
Windows Firewall
Check firewall status
Allow a program through Windows Firev
System
View amount of RAM and processor spe

FIGURE D-4 System and Security window

4. Select the option `System` shown on the right side. A window similar to the window shown in Figure D-5 appears.



View basic information about your computer

Windows edition

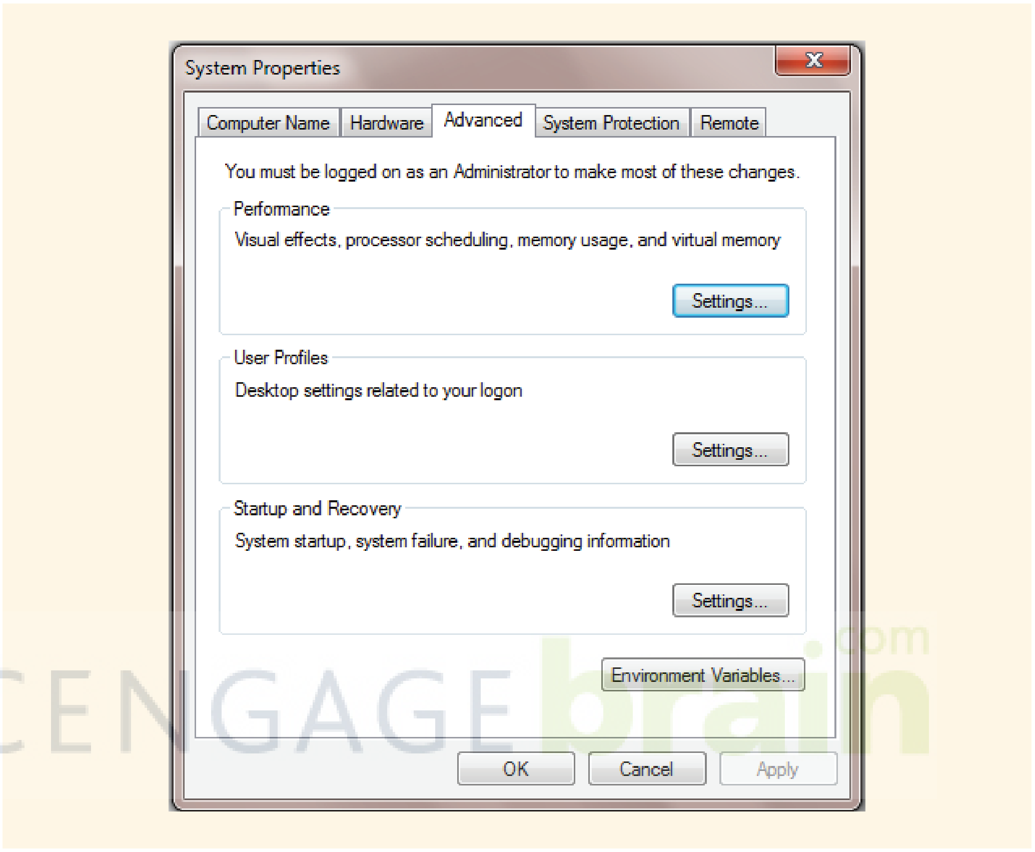
Windows 7 Professional

Copyright © 2009 Microsoft Corporation. All rights reserved.

Get more features with a new edition of Windows 7

FIGURE D-5 System window

5. Select the option `Advanced system settings` shown on the left side. A window similar to the window shown in Figure D-6 appears.



You must be logged on as an Administrator to make most of these changes.

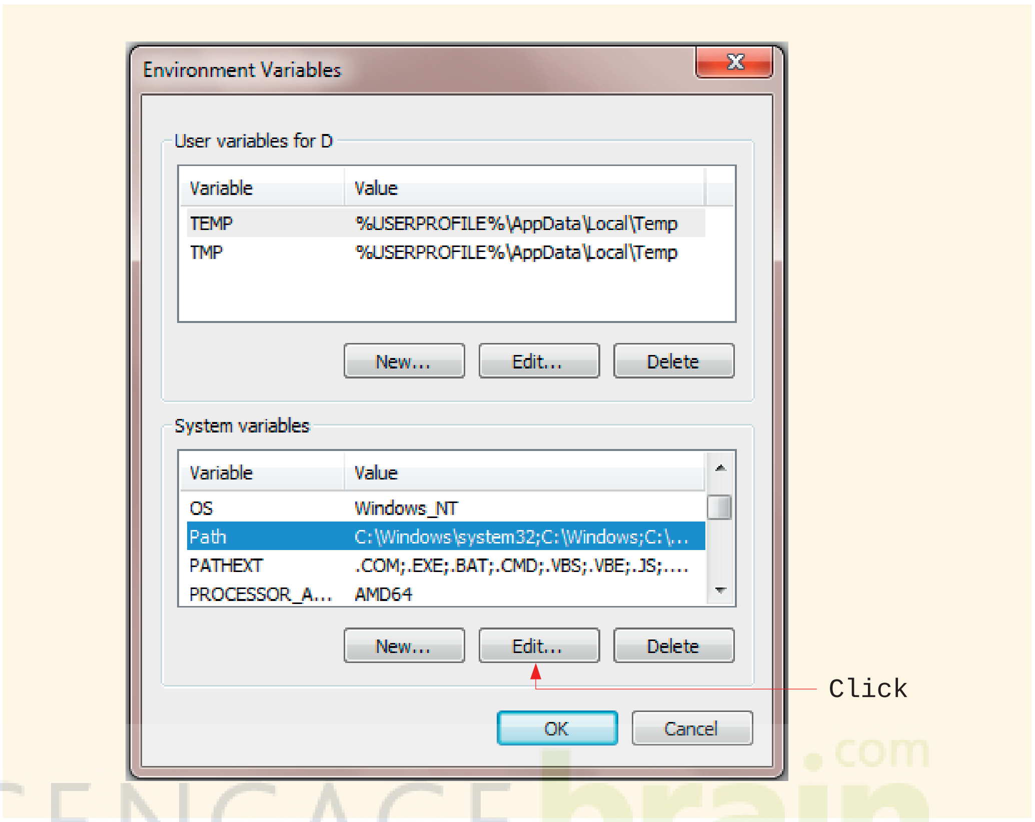
Performance
Visual effects, processor scheduling, memory usage, and virtual memory

User Profiles
Desktop settings related to your logon

Startup and Recovery
System startup, system failure, and debugging information

FIGURE D-6 Selecting the `Advanced` tab

6. In the window in Figure D-6, click `Environment Variables`. A window similar to the one shown in Figure D-7 appears. In this window, in the `System variables` section, scroll down, select `Path`, and then click `Edit`.



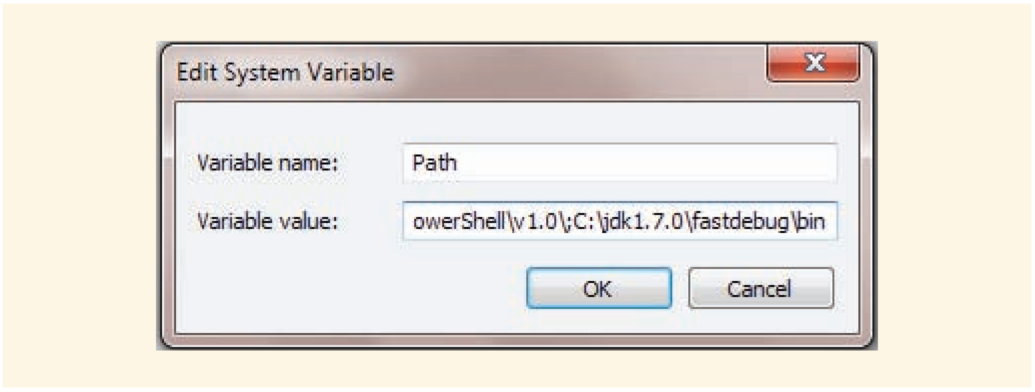
Variable	Value
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

Variable	Value
OS	Windows_NT
Path	C:\Windows\system32;C:\Windows;C:\...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....
PROCESSOR_A...	AMD64

FIGURE D-7 Selecting Path in System variables

7. After you select `Edit`, the window in Figure D-8 appears. In the box following `Variable value:`, type the following and then click `OK` three times. (If the path is different on your system, type in the path as it corresponds to your own installation. For example, the path on your system might be `C:\Program Files\Java\jdk1.7.0\bin`)

   `;C:\jdk1.7.0\fastdebug\bin`



Variable name: Path

Variable value: owerShell\v1.0\;C:\jdk1.7.0\fastdebug\bin

FIGURE D-8 Editing Path

The preceding steps should set the `Path`. To be absolutely certain about the `Path` and also to set the `CLASSPATH`, check your operating system's documentation.

Executing Java Programs

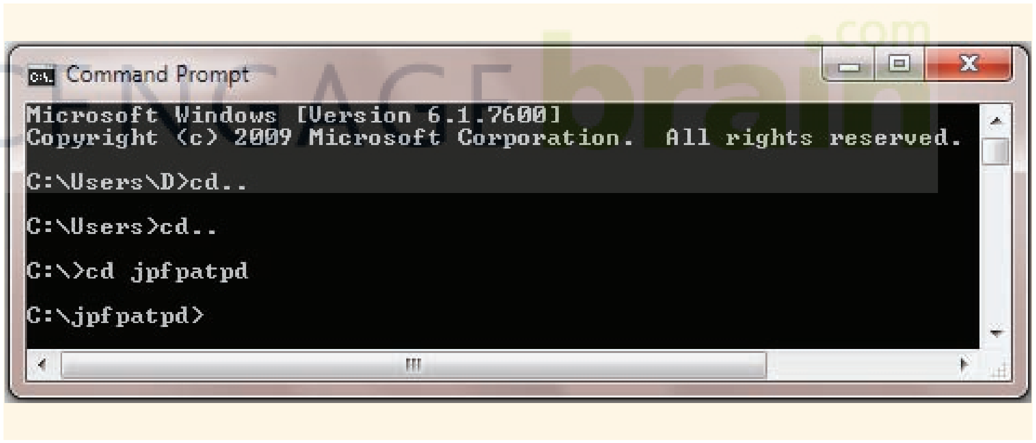
The following discussion assumes that you have set the `Path` so that the files `javac.exe` and `java.exe` can be executed from within any subdirectory.

You can use an editor such as Notepad to create Java programs. The name of the class containing the Java program and the name of the file containing the program must be the same. Moreover, the file containing the Java program must have the extension `.java`.

Suppose that the file `Welcome.java` is in the subdirectory `c:\jpfpatpd` and contains the following Java application program:

```
public class Welcome
{
    public static void main(String[] args)
    {
        System.out.println("Welcome to Java Programming.");
    }
}
```

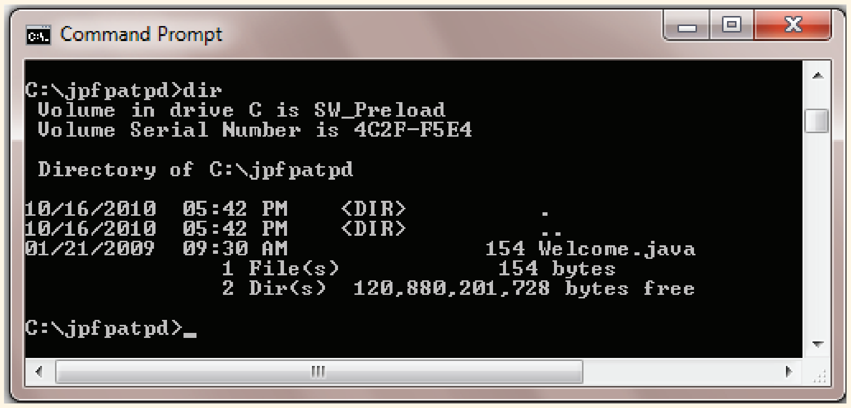
We assume that you have switched to the subdirectory `c:\jpfpatpd` (see Figure D-9).



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\D>cd..

C:\Users>cd..

C:\>cd jpfpatpd

C:\jpfpatpd>
```

FIGURE D-9 Windows console environment

Figure D-10 shows the files in the subdirectory `c:\jpfpatpd`.

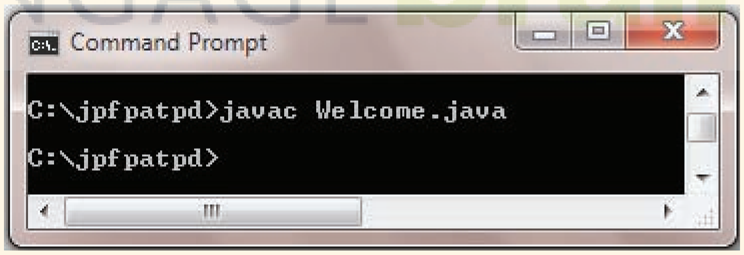


```
C:\jpfpatpd>dir
 Volume in drive C is SW_Preload
 Volume Serial Number is 4C2F-F5E4

 Directory of C:\jpfpatpd

10/16/2010  05:42 PM    <DIR>          .
10/16/2010  05:42 PM    <DIR>          ..
01/21/2009  09:30 AM               154 Welcome.java
               1 File(s)            154 bytes
               2 Dir(s)  120,880,201,728 bytes free

C:\jpfpatpd>_
```

FIGURE D-10 Files in the subdirectory `c:\jpfpatpd`

To place the compiled code of the program `Welcome.java` in the subdirectory `c:\jpfpatpd`, you can execute the following command, as shown in Figure D-11:

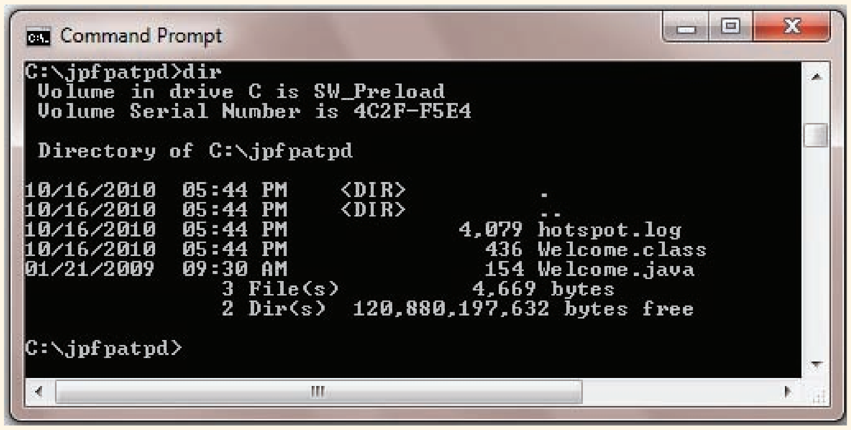
```
javac Welcome.java
```



```
C:\jpfpatpd>javac Welcome.java

C:\jpfpatpd>
```

FIGURE D-11 Compile `Welcome.java` program

The preceding command creates the file `Welcome.class`, which contains the compiled code of the program `Welcome.java` and places it in the subdirectory `c:\jpfpatpd` (see Figure D-12).

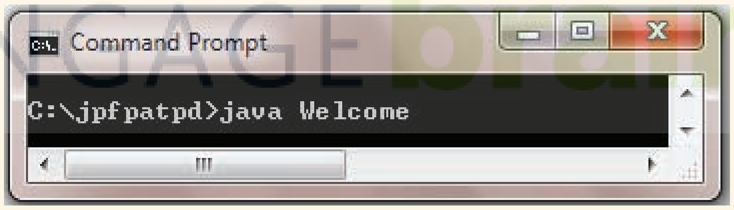


```
C:\jpfpatpd>dir
 Volume in drive C is SW_Preload
 Volume Serial Number is 4C2F-F5E4

 Directory of C:\jpfpatpd

10/16/2010  05:44 PM    <DIR>          .
10/16/2010  05:44 PM    <DIR>          ..
10/16/2010  05:44 PM             4,079 hotspot.log
10/16/2010  05:44 PM               436 Welcome.class
01/21/2009  09:30 AM               154 Welcome.java
               3 File(s)          4,669 bytes
               2 Dir(s)  120,880,197,632 bytes free

C:\jpfpatpd>
```

FIGURE D-12 The file `Welcome.class` program

You can now issue the following command to execute the `Welcome` program (see Figure D-13):

```
java Welcome
```

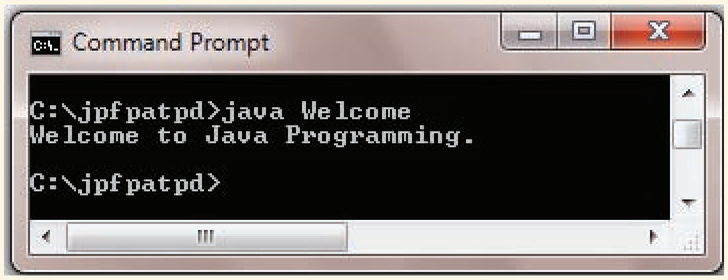


```
C:\jpfpatpd>java Welcome
```

FIGURE D-13 Executing `Welcome` program

After this statement executes, the following line appears on the screen, as shown in Figure D-14:

```
Welcome to Java Programming.
```



```
C:\jpfpatpd>java Welcome
Welcome to Java Programming.

C:\jpfpatpd>
```

FIGURE D-14 Execution of the `Welcome` program

The preceding command, after compiling the program, places the compiled code in the same subdirectory as the program. However, when you compile a Java program using the command-line compiler, you can instruct the system to store the program's compiled code in any subdirectory you want. To place the compiled code in a specific directory, you include the option `–d` and the name of the subdirectory where you want the command code placed when you compile the program. For example, the command:

```
javac -d c:\jdk1.7.0\fastdebug\bin\classes Welcome.java
```

places the compiled code of the program Welcome.java in the subdirectory:

```
c:\jdk1.7.0\fastdebug\bin\classes
```

Note that the subdirectory `c:\jdk1.7.0\fastdebug\bin\classes` must exist before you execute the command to compile the program.

Similarly, the following command places the compiled code of the program `Welcome.java` in the subdirectory `c:\jpfpatpd`:

```
javac -d c:\jpfpatpd Welcome.java
```

To be absolutely certain that the directory path is correct, check your system's documentation.

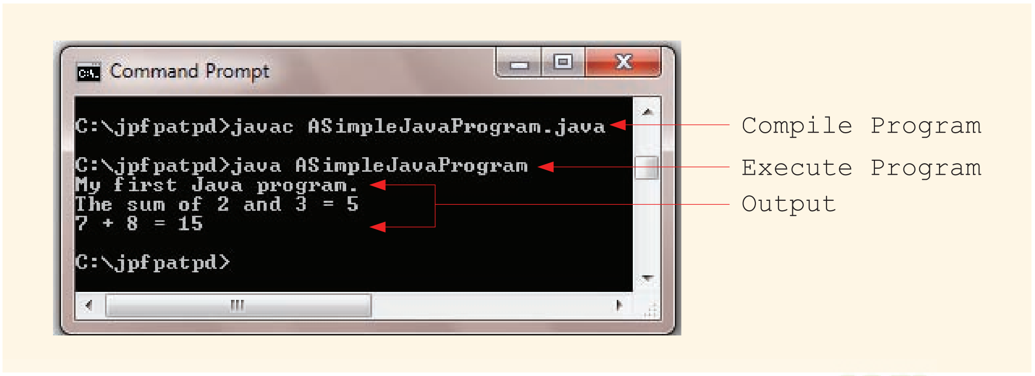
Suppose that you have placed the file `Welcome.class` within the subdirectory `c:\jdk1.7.0\fastdebug\bin\classes`. In addition, suppose that you have not set the `CLASSPATH` to allow the system to look for the compiled code on specific locations on your computer. In this case, you can use the option `-classpath` and the name of the subdirectory that contains the compiled code to execute the program. For example, the following command looks for the compiled code of the `Welcome` program in the subdirectory `c:\jdk1.7.0\fastdebug\bin\classes`:

```
java -classpath c:\jdk1.7.0\fastdebug\bin\classes Welcome
```

NOTE

If the compiled code of the classes is in the subdirectory, say c:\jpfpatpd, you can set the system variable CLASSPATH to c:\jpfpatpd. If the system variable CLASSPATH already exists, you can add the path c:\jpfpatpd to it. To be absolutely certain how to set CLASSPATH in the Windows environment, check your operating system's documentation. Moreover, if you are using other operating systems, such as UNIX, check the documentation to set the variables so that you can conveniently compile and execute a Java program.

The subdirectory c:\jpfpatpd also contains the file ASimpleJavaProgram.java. Figure D-15 shows the compile command, execute command, and the output of the program.

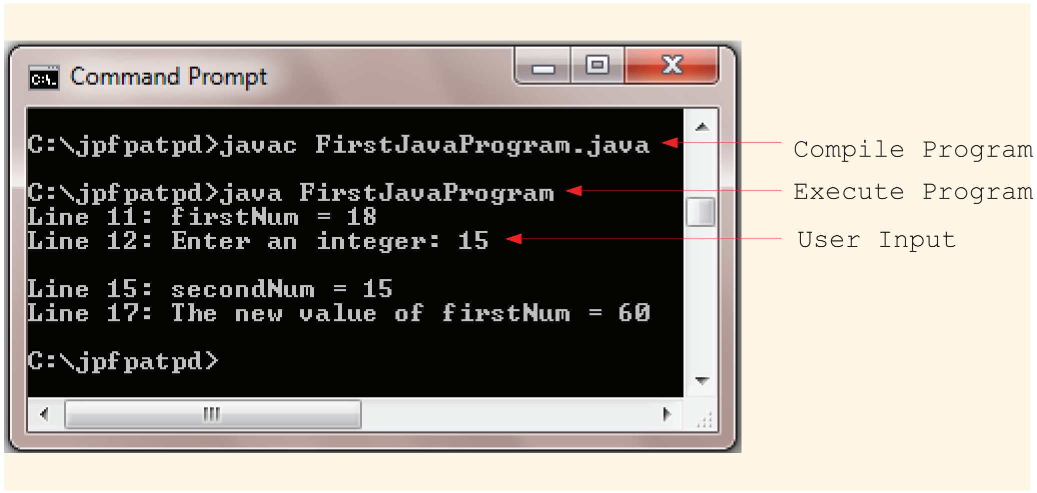


```
C:\jpfpatpd>javac ASimpleJavaProgram.java

C:\jpfpatpd>java ASimpleJavaProgram
My first Java program.
The sum of 2 and 3 = 5
7 + 8 = 15

C:\jpfpatpd>
```

FIGURE D-15 Compiling and executing the program ASimpleJavaProgram.java

Note that the program ASimpleJavaProgram is the same as that discussed in Chapter 2.

The subdirectory c:\jpfpatpd also contains the file FirstJavaProgram.java. Figure D-16 shows the compile command, execute command, and program output.



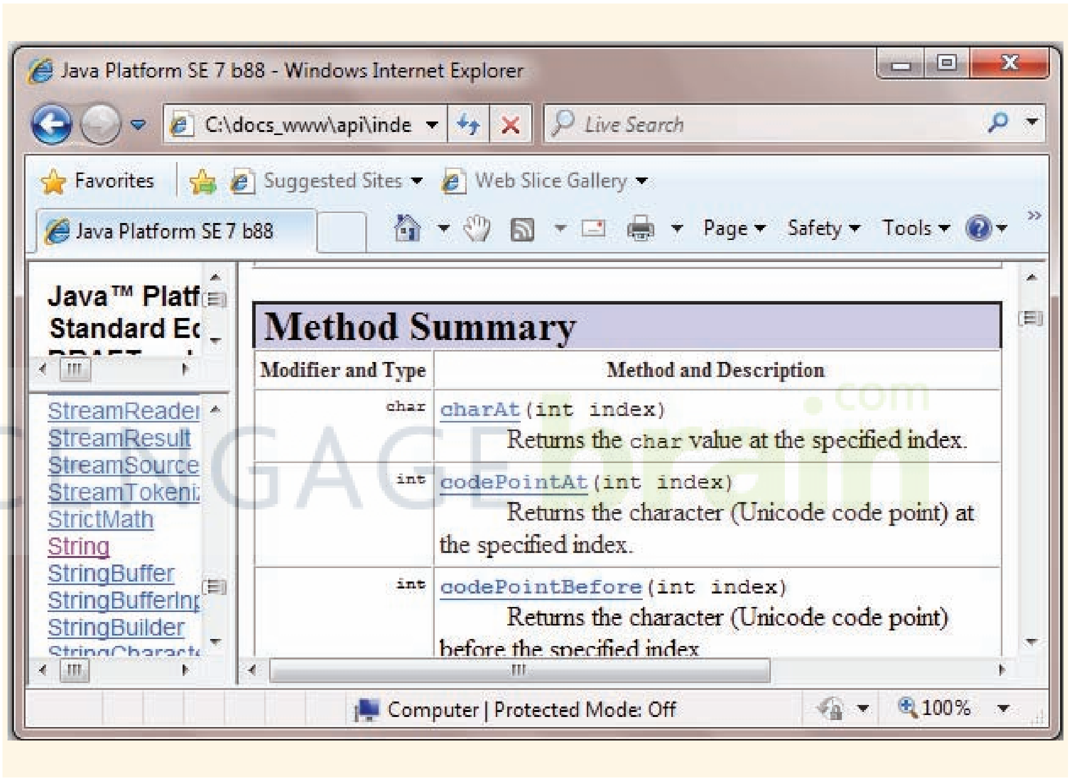
```
C:\jpfpatpd>javac FirstJavaProgram.java

C:\jpfpatpd>java FirstJavaProgram
Line 11: firstNum = 18
Line 12: Enter an integer: 15

Line 15: secondNum = 15
Line 17: The new value of firstNum = 60

C:\jpfpatpd>
```

FIGURE D-16 Compiling and executing the program FirstJavaProgram.java

Note that the program `FirstJavaProgram` is the same as that discussed in Example 2-26 in Chapter 2.

Java Style Documentation

In this book, whenever we designed a class, among others, we provided an explanation of the methods. We also noted that Java provides a wealth of predefined classes. For example, if you visit the Web site *http://java.sun.com/javase/6/docs/api* or *http://java.sun.com/javase/7/docs/api* you can find a description of the **`class`** `String` as shown in Figure D-17. (Note that the URL locations of these api documentations may change without any notice.)



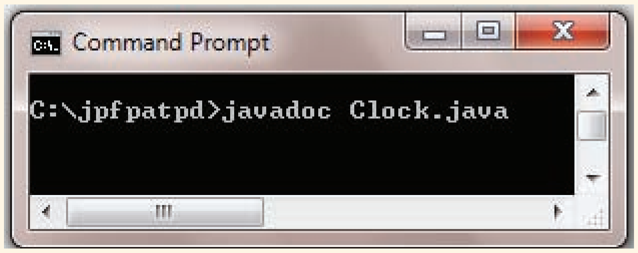
Method Summary

Modifier and Type	Method and Description
char	`charAt(int index)` Returns the char value at the specified index.
int	`codePointAt(int index)` Returns the character (Unicode code point) at the specified index.
int	`codePointBefore(int index)` Returns the character (Unicode code point) before the specified index

FIGURE D-17 The **`class`** `String`

This description of the **`class`** `String` as shown in Figure D-17 is Java style documentation. You can also produce this type of documentation for the classes you design using the command `javadoc`. We illustrate how to produce the Java style documentation of the **`class`** `Clock`, designed in Chapter 8.

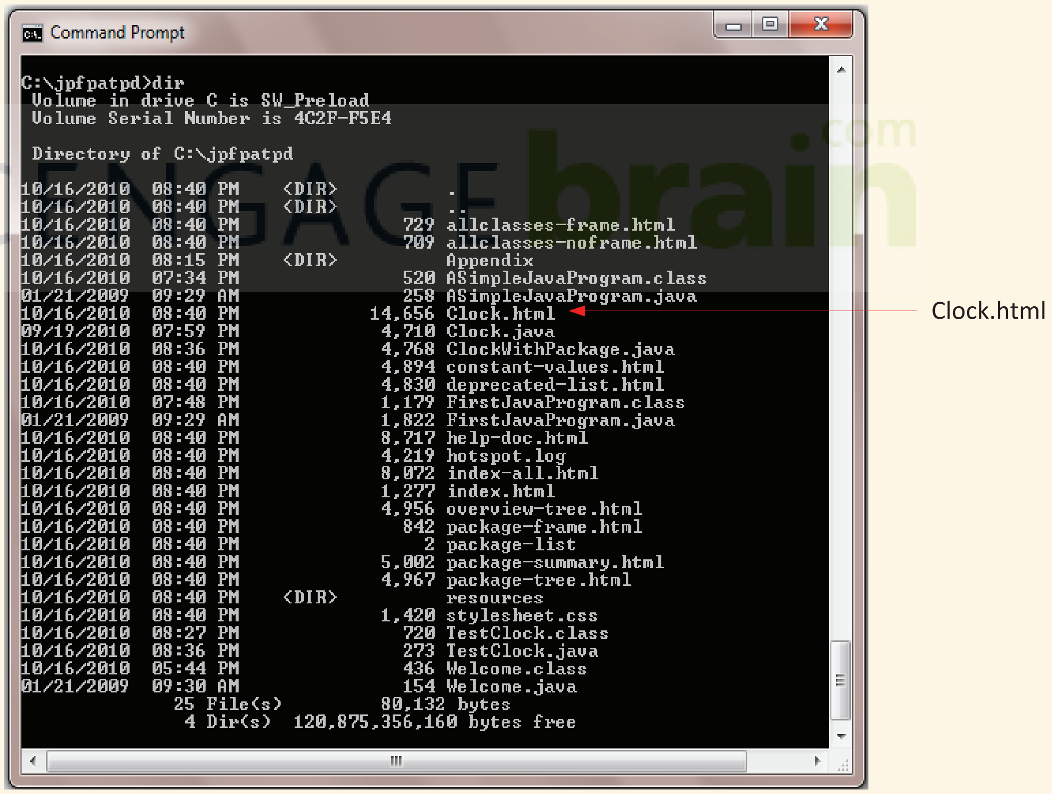
Suppose that the definition of the `class` `Clock` is in the subdirectory `c:\jpfpatpd`. Next execute the command: `javadoc Clock.java`, see Figure D-18.



```
C:\jpfpatpd>javadoc Clock.java
```

FIGURE D-18 Execute the command `javadoc Clock.java`

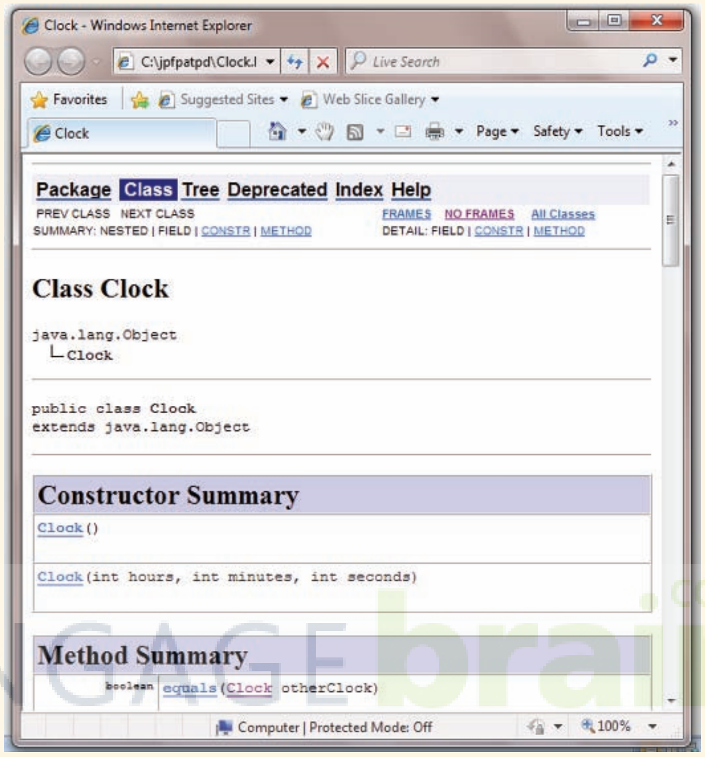
The preceding command creates a number of files as shown in Figure D-19.



```
C:\jpfpatpd>dir
 Volume in drive C is SW_Preload
 Volume Serial Number is 4C2F-F5E4

 Directory of C:\jpfpatpd

10/16/2010  08:40 PM    <DIR>          .
10/16/2010  08:40 PM    <DIR>          ..
10/16/2010  08:40 PM               729 allclasses-frame.html
10/16/2010  08:40 PM               709 allclasses-noframe.html
10/16/2010  08:15 PM    <DIR>          Appendix
10/16/2010  07:34 PM               520 ASimpleJavaProgram.class
01/21/2009  09:29 AM               258 ASimpleJavaProgram.java
10/16/2010  08:40 PM            14,656 Clock.html
09/19/2010  07:59 PM             4,710 Clock.java
10/16/2010  08:36 PM             4,768 ClockWithPackage.java
10/16/2010  08:40 PM             4,894 constant-values.html
10/16/2010  08:40 PM             4,830 deprecated-list.html
10/16/2010  07:48 PM             1,179 FirstJavaProgram.class
01/21/2009  09:29 AM             1,822 FirstJavaProgram.java
10/16/2010  08:40 PM             8,717 help-doc.html
10/16/2010  08:40 PM             4,219 hotspot.log
10/16/2010  08:40 PM             8,072 index-all.html
10/16/2010  08:40 PM             1,277 index.html
10/16/2010  08:40 PM             4,956 overview-tree.html
10/16/2010  08:40 PM               842 package-frame.html
10/16/2010  08:40 PM                 2 package-list
10/16/2010  08:40 PM             5,002 package-summary.html
10/16/2010  08:40 PM             4,967 package-tree.html
10/16/2010  08:40 PM    <DIR>          resources
10/16/2010  08:40 PM             1,420 stylesheet.css
10/16/2010  08:27 PM               720 TestClock.class
10/16/2010  08:36 PM               273 TestClock.java
10/16/2010  05:44 PM               436 Welcome.class
01/21/2009  09:30 AM               154 Welcome.java
              25 File(s)         80,132 bytes
               4 Dir(s)  120,875,356,160 bytes free
```

FIGURE D-19 The files produced by the command `javadoc Clock.java`

Next, if you switch to the Windows environment and double click on the file `Clock.html`, it shows you the Java style documentation of the **`class`** `Clock` shown in Figure D-20.



Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class Clock

```
java.lang.Object
  └Clock
```

```
public class Clock
extends java.lang.Object
```

Constructor Summary

```
Clock()
Clock(int hours, int minutes, int seconds)
```

Method Summary

```
boolean equals(Clock otherClock)
```

FIGURE D-20 Java style documentation of the **`class`** `Clock`

Creating Your Own Packages

Recall that a package is a collection of related classes. As you develop classes, you can create packages and categorize your classes. You can import your classes in the same way that you import classes from the packages provided by Java.

To create a package and add a class to the package so that the class can be used in a program, you do the following:

1. Define the class to be **`public`**. If the class is not **`public`**, it can be used only within the package.
2. Choose a name for the package. To organize your package, you can create subdirectories within the directory that contains the compiled code of the classes. For

example, you could create a directory for the classes you create in this book. Because the title of this book is *Java Programming: From Problem Analysis to Program Design*, you could create a directory named jpfpatpd. You could then make subdirectories for the classes used in each chapter, such as the subdirectory Appendix within the directory jpfpatpd.

Suppose that you want to create a **package** to group the classes related to time. You could call this **package** clockPackage. To add the **class** Clock to this package and to place the package clockPackage within the subdirectory Appendix of the directory jpfpatpd, you include the following package statement with the file containing the **class** Clock (Note that the **class** Clock is the same as discussed in Chapter 8):

```
package jpfpatpd.Appendix.clockPackage;
```

We put this statement before the definition of the **class**, like this:

```
package jpfpatpd.Appendix.clockPackage;

public class Clock
{
    //put instance variables and methods here
}
```

The next step is to compile the file Clock.java using the compile command in the IDE (integrated development environment) you are using.

The following discussion assumes that you have set the Path so that the files javac.exe and java.exe can be executed from within any subdirectory. Suppose that the file Clock.java is in the subdirectory c:\jpfpatpd. We assume that you have switched to the subdirectory c:\jpfpatpd.

If you are using Java 7.0, which contains a command-line compiler, you include the option -d to place the compiled code of the program Clock.java in a specific directory. For example, the command:

```
javac -d c:\jre1.7.0\lib\classes Clock.java
```

places the compiled code of the program Clock.java in the subdirectory:

```
c:\jre1.7.0\lib\classes\jpfpatpd\Appendix\clockPackage
```

Similarly, the following command places the compiled code of the program Clock.java in the subdirectory c:\jpfpatpd\Appendix\clockPackage:

```
javac -d c:\ Clock.java
```

If the directories jpfpatpd, Appendix, and clockPackage do not exist, then the compiler automatically creates these directories. Note that for the earlier command to execute successfully, the subdirectory c:\jre1.7.0\lib\classes must exist. If this subdirectory does not exist, you must first create it. Also, to be absolutely sure about the correct directory path, check your system's documentation. Moreover, if you do not use the -d option with the path of the subdirectory to specify the subdirectory in which to store the compiled code, then the compiled code is, typically, stored in the current subdirectory.

Once the package is created, you can use the appropriate import command in your program to make use of the class. For example, to use the class Clock, as created in the preceding code, you use the following import statement in your program:

```
import jpfpatpd.Appendix.clockPackage.Clock;
```

In Java, package is a reserved word.

Example D-1 further explains how to use a package in a program. We assume that the class Clock has been compiled and placed in the subdirectory c:\jpfpatpd\Appendix\clockPackage.

EXAMPLE D-1

The following program uses the class Clock.

```
import jpfpatpd.Appendix.clockPackage.Clock;

public class TestClock
{
    public static void main(String[] args)
    {
        Clock myClock = new Clock(12,30,45);

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

Because this program uses the class Clock, when you compile the program using the compiler command, you use the option -classpath to specify where to find the compiled code of the class Clock. Suppose that the file TestClock.java is in the subdirectory c:\jpfpatpd. Consider the following command:

```
javac -classpath c:\ TestClock.java
```

This command finds Clock.class in the subdirectory c:\jpfpatpd\Appendix\clockPackage. The compiled code TestClock.class of the program TestClock.java is placed in the current subdirectory. On the other hand, the following command places the compiled code, TestClock.class, in the subdirectory c:\jpfpatpd:

```
javac -d c:\jpfpatpd -classpath c:\ TestClock.java
```

Suppose the file TestClock.class is in the subdirectory c:\jpfpatpd. The following command executes the program TestClock.class:

```
java -classpath .;c:\ TestClock
```

If you are using an IDE to create Java programs, you need to be familiar with the commands to compile and execute them. Typically, an IDE automatically stores the compiled code of the classes in an appropriate subdirectory.

Multiple-File Programs

In the preceding section, you learned how to create a `package`. Creating a `package` to group related `class`(es) is very useful if the classes are to be used again and again. On the other hand, if a `class` is to be used in only one program, or if you have divided your program so that it uses more than one `class`, rather than create a `package`, you can directly add the file(s) containing the `class`(es) to the program.

The Java IDEs—J++ Builder and JGrasp—put the editor, compiler, and loader all into one program. With one command, a program is compiled. These IDEs also manage multiple-file programs in the form of a project. A **project** consists of several files, called the project files. These IDEs include a command that allows you to add several files to a project. Also, these IDEs usually have commands such as **build**, **rebuild**, or **make** (check your software's documentation) to automatically compile all the files required. When one or more files in the project change, you can use these commands to recompile the files.

Formatting the Output of Decimal Numbers Using the **class DecimalFormat**

Chapter 3 explained how to format the output of floating-point numbers, using the method format of the `class String`, to a specific number of decimal places. Chapter 3 also noted that another way to format the output of floating-point numbers is to use the `class DecimalFormat`.

Recall that the default output of decimal numbers of the type `float` is up to six decimal places. Similarly, the default output of decimal numbers of the type `double` is up to 15 decimal places. For example, consider the statements in Table D-3; the output is shown to the right.

TABLE D-3 Default Output of Floating-Point Numbers

Statement	Output
`System.out.println(22.0 / 7.0);`	`3.142857142857143`
`System.out.println(75.0 / 7.0);`	`10.714285714285714`
`System.out.println((float)(33.0 / 16.0));`	`2.0625`
`System.out.println((float)(22.0 / 7.0));`	`3.142857`

As discussed in Chapter 3, sometimes floating-point numbers must be output in a specific way. For example, a paycheck must be printed to two decimal places, whereas the results of a scientific experiment might require the output of floating-point numbers to six, seven, or perhaps even 10 decimal places.

You can use the Java `class DecimalFormat` to format decimal numbers in a specific manner. The method `format` of the `class DecimalFormat` is applied to the decimal value being formatted. The following steps explain how to use these features to format decimal numbers:

1. Create a `DecimalFormat` object and initialize it to the specific format. Consider the following statement:

   ```
   DecimalFormat twoDecimal = new DecimalFormat("0.00");
   ```

   This statement creates the `DecimalFormat` object `twoDecimal` and initializes it to the string `"0.00"`. Each `0` in the string is a **format flag**. The string `"0.00"` specifies the formatting of the decimal number. This string indicates that the decimal number being formatted with the object `twoDecimal` will have at least one digit to the left of the decimal point and exactly two digits to the right of the decimal point. If the number being formatted does not meet the formatting requirement, that is, it does not have digits at the specified places, those places are automatically filled with 0. Moreover, suppose that you have the following statement:

   ```
   DecimalFormat twoDigits = new DecimalFormat("0.##");
   ```

   The object `twoDigits` can be used to format the number with two decimal places, but the `##` symbols indicate that trailing zeros will appear as spaces.
2. Next, use the method `format` of the `class DecimalFormat`. (Assume the first declaration of Step 1.) For example, the statement:

   ```
   twoDecimal.format(56.379);
   ```

   formats the decimal number `56.379` as `56.38` (the decimal number is rounded). The method format returns the string containing the digits of the formatted number.
3. The `class DecimalFormat` is included in the `package java.text`. You must import this `class` into your program.

Example D-2 illustrates how to format the output of decimal numbers.

EXAMPLE D-2

```
//Program: Formatting output of decimal numbers using
//the class DecimalFormat

import java.text.DecimalFormat;

public class FormattingDecimalNum
{
    public static void main(String[] args)
    {
        double x = 15.674;                              //Line 1
        double y = 235.73;                              //Line 2
        double z = 9525.9864;                           //Line 3
```

```
        DecimalFormat twoDecimal =
                       new DecimalFormat("0.00");     //Line 4
        DecimalFormat threeDecimal =
                         new DecimalFormat("0.000");  //Line 5

        System.out.println("Line 6: Outputting the "
                         + "values of x, y, and z \n"
                         + "        with two decimal "
                         + "places.");                //Line 6
        System.out.println("Line 7: x = "
                         + twoDecimal.format(x));     //Line 7
        System.out.println("Line 8: y = "
                         + twoDecimal.format(y));     //Line 8
        System.out.println("Line 9: z = "
                         + twoDecimal.format(z));     //Line 9

        System.out.println("Line 10: Outputting the "
                         + "values of x, y, and z \n"
                         + "         with three "
                         + "decimal places.");        //Line 10
        System.out.println("Line 11: x = "
                         + threeDecimal.format(x));   //Line 11
        System.out.println("Line 12: y = "
                         + threeDecimal.format(y));   //Line 12
        System.out.println("Line 13: z = "
                         + threeDecimal.format(z));   //Line 13
    }
}
```

Sample Run:

```
Line 6: Outputting the values of x, y, and z
        with two decimal places.
Line 7: x = 15.67
Line 8: y = 235.73
Line 9: z = 9525.99
Line 10: Outputting the values of x, y, and z
         with three decimal places.
Line 11: x = 15.674
Line 12: y = 235.730
Line 13: z = 9525.986
```

The statements in Lines 1, 2, and 3 declare and initialize `x`, `y`, and `z` to `15.674`, `235.73`, and `9525.9864`, respectively. The statement in Line 4 creates and initializes the `DecimalFormat` object `twoDecimal` to output decimal numbers to two decimal places. Similarly, the statement in Line 5 creates and initializes the `DecimalFormat` object `threeDecimal` to output decimal numbers with three decimal places.

The statements in Lines 7, 8, and 9 output the values of `x`, `y`, and `z` to two decimal places, respectively. Note that the printed values of `x` in Line 7 and `z` in Line 9 are rounded.

The statements in Lines 11, 12, and 13 output the values of `x`, `y`, and `z`, respectively, to three decimal places. Note that the value of `y` in Line 12 is output to three decimal places. Because the number stored in `y` has only two decimal places, a `0` is printed as the third decimal place.

Packages and User-Defined Classes

Chapter 7 discusses user-defined methods, in particular methods with parameters. As explained in Chapter 3, there are two types of variables in Java—primitive and reference. The program in Example 7-8 illustrates that if a formal parameter is of the primitive type and the corresponding actual parameter is a variable, then the formal parameter cannot change the value of the actual parameter. Changing the value of a formal parameter of a primitive data type has no effect on the actual parameter. However, if a formal parameter is a reference variable, then both the actual and the formal parameter refer to the same object. That is, only formal parameters that are reference variables are capable of passing values outside the function.

Java provides classes corresponding to each primitive data type, so that values of primitive data types can be considered objects. For example, you can use the `class Integer` to treat `int` values as objects, `class Double` to treat `double` values as objects, and so on. These classes, called wrapper classes, were described in Chapter 6.

As noted in Chapter 7, Java does not provide any class that wraps primitive type values in objects and, when passed as parameters, change their values. If a method returns only one value of a primitive type, then you can write a value-returning method. However, if you encounter a situation that requires you to write a method that needs to pass more than one value of a primitive type, then you should design your own classes. In the next section, we introduce various classes to accomplish this. For example, we design the `class IntClass` so that values of the `int` type can be wrapped in an object. The `class IntClass` also provides methods to change the value of an `IntClass` object. We use reference variables of the `IntClass` type to pass `int` values outside a method.

Primitive Type Classes

This section presents the definitions of the `class`es `IntClass`, `LongClass`, `CharClass`, `FloatClass`, `DoubleClass`, and `BooleanClass`.

Class: IntClass

```
public class IntClass
{
    private int x;    //variable to store the number

        //default constructor
        //Postcondition: x = 0
    public IntClass()
    {
        x = 0;
    }
```

```
        //constructor with parameter
        //Postcondition: x = num
    public IntClass(int num)
    {
        x = num;
    }

       //Method to set the data member x
       //Postcondition: x = num
    public void setNum(int num)
    {
        x = num;
    }

       //Method to return the value of x
       //Postcondition: The value of x is returned
    public int getNum()
    {
        return x;
    }

        //Method to update the value of x by adding
        //the value of num
        //Postcondition: x = x + num;
    public void addToNum(int num)
    {
        x = x + num;
    }
        //Method to update the value of x by multiplying
        //the value of x by num
        //Postcondition: x = x * num;
    public void multiplyToNum(int num)
    {
        x = x * num;
    }

        //Method to compare the value of x with the value of num
        //Postcondition: Returns a value < 0 if x < num
        //               Returns 0 if x == num
        //               Returns a value > 0 if x > num
    public int compareTo(int num)
    {
        return (x - num);
    }

        //Method to compare x with num for equality
        //Postcondition: Returns true if x == num;
        //               otherwise it returns false
    public boolean equals(int num)
    {
        if (x == num)
            return true;
        else
            return false;
    }
```

```
        //Method to return the value of x as a string
    public String toString()
    {
        return (String.valueOf(x));
    }
}
```

Consider the following statements:

```
IntClass firstNum = new IntClass();       //Line 1
IntClass secondNum = new IntClass(5);     //Line 2
int num;                                  //Line 3
```

The statement in Line 1 creates the object `firstNum` and initializes it to `0`. The statement in Line 2 creates the object `secondNum` and initializes it to `5`. The statement in Line 3 declares `num` to be an `int` variable. Now consider the following statements:

```
firstNum.setNum(24);                      //Line 4
secondNum.addToNum(6);                    //Line 5
num = firstNum.getNum();                  //Line 6
```

The statement in Line 4 sets the value of `firstNum` (in fact, the value of the data member `x` of `firstNum`) to `24`. The statement in Line 5 updates the value of `secondNum` to `11` (the previous value `5` is updated by adding `6` to it.) The statement in Line 6 retrieves the value of the object `firstNum` (the value of the data member `x`) and assigns it to `num`. After this statement executes, the value of `num` is `24`.

The following statements output the values of `firstNum` and `secondNum` (in fact, the values of their data members):

```
System.out.println("firstNum = " + firstNum);
System.out.println("secondNum = " + secondNum);
```

Table D-4 shows how variables of `int` type and the corresponding reference variables of `IntClass` type work.

TABLE D-4 Variables of `int` Type and the Corresponding Reference Variables of `IntClass`

	`int`	`IntClass`

Declaration without or with initialization	`int x, y = 5;`	`IntClass x, y;`
		`x = new IntClass();`
		`y = new IntClass(5);`

Assignment	`x = 24;`	`x.setNum(24);`
	`y = x;`	`y.setNum(x.getNum());`
Addition	`x = x + 10;`	`x.addToNum(10);`
	`x = x + y;`	`x.addToNum(y.getNum());`

TABLE D-4 Variables of `int` Type and the Corresponding Reference Variables of `IntClass` (continued)

	`int`	`IntClass`
Multiplication	`x = x * 10;`	`x.multiplyToNum(10);`
	`x = x * y;`	`x.multiplyToNum(y.getNum());`
Comparison	`if (x < 10)`	`if (x.compareTo(10) < 0)`
	`if (x < y)`	`if (x.compareTo(y.getNum()) < 0)`
	`if (x <= 10)`	`if (x.compareTo(10) <= 0)`
	`if (x <= y)`	`if (x.compareTo(y.getNum()) <= 0)`
	`if (x == 10)`	`if (x.compareTo(10) == 0)` or `if (x.equals(10))`
	`if (x == y)`	`if (x.compareTo(y.getNum()) == 0)` or `if (x.equals(y.getNum()))`
	`if (x > 10)`	`if (x.compareTo(10) > 0)`
	`if (x > y)`	`if (x.compareTo(y.getNum()) > 0)`
	`if (x >= 10)`	`if (x.compareTo(10) >= 0)`
	`if (x >= y)`	`if (x.compareTo(y.getNum()) >= 0)`
	`if (x != 10)`	`if (x.compareTo(10) != 0)` or `if (!x.equals(10))`
	`if (x != y)`	`if (x.compareTo(y.getNum()) != 0)` or `if (!x.equals(y.getNum()))`
Output		`System.out.println(x);`
		`System.out.println(x);`

Class: LongClass

```
public class LongClass
{
    private long x;

    public LongClass()
    {
        x = 0;
    }

    public LongClass(long num)
    {
        x = num;
    }

    public void setNum(long num)
    {
        x = num;
    }

    public long getNum()
    {
        return x;
    }

    public void addToNum(long num)
    {
        x = x + num;
    }

    public void multiplyToNum(long num)
    {
        x = x * num;
    }

    public long compareTo(long num)
    {
        return (x - num);
    }

    public boolean equals(long num)
    {
        if (x == num)
            return true;
        else
            return false;
    }

    public String toString()
    {
        return (String.valueOf(x));
    }
}
```

Class: CharClass

```
public class CharClass
{
    private char ch;

    public CharClass()
    {
        ch = ' ';
    }

    public CharClass(char c)
    {
        ch = c;
    }

    public void setChar(char c)
    {
        ch = c;
    }

    public int getChar()
    {
        return ch;
    }

    public char nextChar()
    {
        return (char)((int)ch + 1);
    }

    public char prevChar()
    {
        return (char)((int)ch - 1);
    }

    public String toString()
    {
        return (String.valueOf(ch));
    }
}
```

Class: FloatClass

```
public class FloatClass
{
    private float x;

    public FloatClass()
    {
        x = 0;
    }

    public FloatClass(float num)
    {
        x = num;
    }
    public void setNum(float num)
    {
        x = num;
    }

    public float getNum()
    {
        return x;
    }

    public void addToNum(float num)
    {
        x = x + num;
    }

    public void multiplyToNum(float num)
    {
        x = x * num;
    }

    public float compareTo(float num)
    {
        return (x - num);
    }

    public boolean equals(float num)
    {
        if (x == num)
            return true;
        else
            return false;
    }

    public String toString()
    {
        return (String.valueOf(x));
    }
}
```

Class: DoubleClass

```
public class DoubleClass
{
    private double x;

    public DoubleClass()
    {
        x = 0;
    }

    public DoubleClass(double num)
    {
        x = num;
    }
    public void setNum(double num)
    {
        x = num;
    }

    public double getNum()
    {
        return x;
    }

    public void addToNum(double num)
    {
        x = x + num;
    }

    public void multiplyToNum(double num)
    {
        x = x * num;
    }

    public double compareTo(double num)
    {
        return (x - num);
    }

    public boolean equals(double num)
    {
        if (x == num)
            return true;
        else
            return false;
    }
    public String toString()
    {
        return (String.valueOf(x));
    }
}
```

Class: BooleanClass

```
public class BooleanClass
{
    private boolean flag;

    public BooleanClass()
    {
        flag = false;
    }

    public BooleanClass(boolean f)
    {
        flag = f;
    }
    public boolean get()
    {
        return flag;
    }

    public void set(boolean f)
    {
        flag = f;
    }

    public String toString()
    {
        return (String.valueOf(flag));
    }

}
```

Using Primitive Type Classes in a Program

This section describes how to use the classes introduced in the previous section.

The class IntClass can be used in two ways. One way is to keep the file IntClass.java and the program in the same directory. First, compile the file IntClass.java, then compile the program.

The second way is to first create a package, and then put this class in that package. For example, you can create the package:

```
jpfpatpd.ch07.primitiveTypeClasses
```

and put the class in this package.

In this case, you place the statement:

```
package jpfpatpd.ch07.primitiveTypeClasses;
```

before the definition of the class IntClass.

The class IntClass definition is in the file IntClass.java. We need to compile this file and place the compiled code in the directory: jpfpatpd.ch07. primitiveTypeClasses. To do so, we execute the following command at the command line:

```
javac -d c:\jre1.7.0\lib\classes IntClass.java
```

The file IntClass.class is now placed in the subdirectory jpfpatpd\ch07 \primitiveTypeClasses of the directory c:\jre1.7.0 \lib\classes.

On the other hand, the command:

```
javac IntClass.java
```

places the file IntClass.class in the subdirectory jpfpatpd\ch07\primitive TypeClasses of the same directory. Note that the system automatically creates the subdirectory jpfpatpd\ch07\primitiveTypeClasses if it does not exist.

You can now import this class in a program using the import statement. For example, you can use either of the following statements to use the `class IntClass` in your program:

```
import jpfpatpd.ch07.primitiveTypeClasses.*;
```

or

```
import jpfpatpd.ch07.primitiveTypeClasses.IntClass;
```

USING A SOFTWARE DEVELOPMENT KIT (SDK)

If you are using an SDK, such as CodeWarrior or J++ Builder, you can place the file containing the definition of the class in the same directory that contains your program. You do not need to create a package. However, you can also create a package using the SDK. In this case, place the appropriate `package` statement before the definition of the class, and use the compile command provided by the SDK. (In most cases, you do not need to specify a subdirectory.) The compiled file will be placed in the appropriate directory. You can now import the class without adding it to the project.

NOTE If you have created a package for your classes, to avoid compilation errors, do not add the file containing the definition of the class to the project.

Enumeration Types

Chapter 2 defined a data type as a set of values, combined with a set of operations on those values. It then introduced the primitive data types: `int`, `char`, `double`, and `float`. Using primitive data types, Chapter 8 discussed how to design classes to create your own data types. In other words, primitive data types are the building blocks of classes.

The values belonging to primitive data types are predefined. Java allows programmers to create their own data types by specifying the values of that data type. These are called **enumeration** or **enum** types and are defined using the keyword `enum`. *The values that you specify for the data types are identifiers*. For example, consider the following statement:

```
enum Grades {A, B, C, D, F};
```

This statement defines `Grades` to be an `enum` type; the values belonging to this type are `A`, `B`, `C`, `D`, and `F`. The values of an `enum` type are called **enumeration** or **enum** constants. Note that the values are enclosed in braces and separated by commas. Also, the `enum` constants within an `enum` type must be unique.

Similarly, the statement:

```
enum Sports {BASEBALL, BASKETBALL, FOOTBALL, GOLF,
             HOCKEY, SOCCER, TENNIS};
```

defines `Sports` to be an `enum` type and the values belonging to this type, that is, the `enum` constants, are `BASEBALL`, `BASKETBALL`, `FOOTBALL`, `GOLF`, `HOCKEY`, `SOCCER`, and `TENNIS`.

Each `enum` type is a *special type of class*, and the values belonging to the `enum` type are (special types of) objects of that class. For example, `Grades` is, in fact, a class and `A`, `B`, `C`, `D`, and `F` are `public static` reference variables to objects of the type `Grades`.

After an `enum` type is defined, you can declare reference variables of that type. For example, the following statement declares `myGrade` to be a reference variable of the `Grades` type:

```
Grades myGrade;
```

Because each of the variables `A`, `B`, `C`, `D`, and `F` is `public` and `static`, they can be accessed using the name of the class and the dot operator. Therefore, the following statement assigns the object `B` to `myGrade`:

```
myGrade = Grades.B;
```

The output of the statement:

```
System.out.println("myGrade: " + myGrade);
```

is:

```
myGrade: B
```

Similarly, the output of the statement:

```
System.out.println("Grades.B: " + Grades.B);
```

is:

```
Grades.B: B
```

Each `enum` constant in an `enum` type has a specific value, called the **ordinal value**. The ordinal value of the first `enum` constant is `0`, the ordinal value of the second `enum` constant is 1, and so on. Therefore, in the `enum` type `Grades`, the ordinal value of `A` is `0` and the ordinal value of `C` is `2`.

Associated with `enum` type is a set of methods that can be used to work with `enum` types. Table D-5 describes some of those methods.

TABLE D-5 Methods Associated with `enum` Types

Method	Description
`ordinal()`	Returns the ordinal value of an `enum` constant
`name()`	Returns the name of the `enum` value
`values()`	Returns the values of an `enum` type as a list

Example D-3 illustrates how these methods work.

EXAMPLE D-3

```
public class EnumExample1
{
    enum Grades {A, B, C, D, F};                         //Line 1

    enum Sports {BASEBALL, BASKETBALL, FOOTBALL,
                 GOLF, HOCKEY, SOCCER, TENNIS};          //Line 2

    public static void main(String[] args)               //Line 3
    {
        Grades myGrade;                                  //Line 4
        Sports mySport;                                  //Line 5

        myGrade = Grades.A;                              //Line 6

        mySport = Sports.BASKETBALL;                     //Line 7

        System.out.println("Line 8: My grade: "
                         + myGrade);                     //Line 8
        System.out.println("Line 9: The ordinal "
                         + "value of myGrade is "
                         + myGrade.ordinal());           //Line 9
        System.out.println("Line 10: myGrade name: "
                         + myGrade.name());              //Line 10

        System.out.println("Line 11: My sport: "
                         + mySport);                     //Line 11
        System.out.println("Line 12: The ordinal "
                         + "value of mySport is "
                         + mySport.ordinal());           //Line 12
        System.out.println("Line 13: mySport name: "
                          + mySport.name());             //Line 13

        System.out.println("Line 14: Sports: ");         //Line 14

        for (Sports sp : Sports.values())                //Line 15
            System.out.println(sp + "'s ordinal "
                             + "value is "
                             + sp.ordinal());            //Line 16

        System.out.println();                            //Line 17
    }
}
```

Sample Run:

```
Line 8: My grade: A
Line 9: The ordinal value of myGrade is 0
Line 10: myGrade name: A
Line 11: My sport: BASKETBALL
Line 12: The ordinal value of mySport is 1
Line 13: mySport name: BASKETBALL
Line 14: Sports:
BASEBALL's ordinal value is 0
BASKETBALL's ordinal value is 1
FOOTBALL's ordinal value is 2
GOLF's ordinal value is 3
HOCKEY's ordinal value is 4
SOCCER's ordinal value is 5
TENNIS's ordinal value is 6
```

The preceding program works as follows. The statements in Lines 1 and 2 define the **enum** type Grades and Sports, respectively. The statement in Line 4 declares myGrade to be a reference variable of the type Grades, and the statement in Line 5 declares mySport to be a reference variable of the type Sports. The statement in Line 6 assigns the object A to myGrade, and the statement in Line 7 assigns the object BASKETBALL to mySport.

The statement in Line 8 outputs myGrade, the statement in Line 9 uses the method ordinal to output the ordinal value of myGrade, and the statement in Line 10 uses the method name to output the name of myGrade.

The statement in Line 11 outputs mySport, the statement in Line 12 uses the method ordinal to output the ordinal value of mySport, and the statement in Line 13 uses the method name to output the name of mySport.

The foreach loop in Line 15 outputs the value of Sports and their ordinal values. Note that the method values, in the expression Sports.values(), returns the value of the **enum** type Sport as a list. The loop control variable sp ranges over those values one-by-one, starting at the first value.

The beginning of this section noted that an **enum** type is a special type of class, and the **enum** constants are reference variables to the objects of that **enum** type. Because each **enum** type is a class, in addition to the **enum** constants, it can also contain constructors, (**private**) data members, and methods. Before describing enumeration type or **enum** in more detail, let us note the following:

1. Enumeration types are defined using the keyword **enum** rather than **class**.
2. **enum** types are implicitly **final** because **enum** constants should not be modified.
3. **enum** constants are implicitly **static**.

4. Once an **enum** type is created, you can declare reference variables of that type, but you cannot instantiate objects using the operator **new**. In fact, an attempt to instantiate an object using the operator **new** will result in a compilation error.

(Because **enum** objects cannot be instantiated using the operator **new**, the constructor, if any, of an enumeration *cannot* be **public**. In fact, the constructors of an **enum** type are implicitly **private**.)

The **enum** type `Grades` was defined earlier in this section. Let us redefine this **enum** type by adding constructors, data members, and methods. Consider the following definition:

```java
public enum Grades
{
    A ("Range 90% to 100%"),
    B ("Range 80% to 89.99%"),
    C ("Range 70% to 79.99%"),
    D ("Range 60% to 69.99%"),
    F ("Range 0% to 59.99%");

    private final String range;

    private Grades()
    {
        range = "";
    }

    private Grades(String str)
    {
        range = str;
    }

    public String getRange()
    {
        return range;
    }
}
```

This **enum** type `Grades` contains the **enum** constants `A`, `B`, `C`, `D`, and `F`. It has a **private** named constant `range` of the type `String`, two constructors, and the method `getRange`. Note that each `Grades` object has the data member `range`. Let us consider the statement:

```java
A ("Range 90% to 100%")
```

This statement creates the `Grades` object, using the constructor with parameters, with the string `"Range 90% to 100%"`, and assigns that object to the reference variable `A`. The method `getRange` is used to return the string contained in the object.

It is not necessary to specify the modifier **private** in the heading of the constructor. Each constructor is implicitly **private**. Therefore, the two constructors of the **enum** type `Grades` can be written as:

```
Grades()
{
    range = "";
}

Grades(String str)
{
    range = str;
}
```

Example D-4 illustrates how the **enum** type `Grades` works.

EXAMPLE D-4

```
public class EnumExample2
{
    public static void main(String[] args)
    {
        System.out.println("Grade Ranges");     //Line 1

        for (Grades gr : Grades.values())       //Line 2
            System.out.println(gr + " "
                             + gr.getRange()); //Line 3

        System.out.println();                   //Line 4
    }
}
```

Sample Run:

```
Grade Ranges
A Range 90% to 100%
B Range 80% to 89.99%
C Range 70% to 79.99%
D Range 60% to 69.99%
F Range 0% to 59.99%
```

The foreach loop in Line 2 uses the method `values` to retrieve the **enum** constants as a list. The method `getRange` in Line 3 is used to retrieve the string contained in the `Grades` object.

The following programming example uses an **enum** type to create a program to play the game of rock, paper, and scissors.

PROGRAMMING EXAMPLE: The Rock, Paper, and Scissors Game

Everyone is familiar with the rock, paper, and scissors game. The game has two players, each of whom chooses one of the three objects: rock, paper, or scissors. If player 1 chooses rock and player 2 chooses paper, player 2 wins the game because paper covers the rock. The game is played according to the following rules:

- If both players choose the same object, this play is a tie.
- If one player chooses rock and the other chooses scissors, the player choosing the rock wins this play because the rock crushes the scissors.
- If one player chooses rock and the other chooses paper, the player choosing the paper wins this play because the paper covers the rock.
- If one player chooses scissors and the other chooses paper, the player choosing the scissors wins this play because the scissors cut the paper.

We write an interactive program that allows two players to play this game.

Input: This program has two types of input:

- The players' responses to play the game
- The players' choices

Output: The players' choices and the winner of each play. After the game is over, the total number of plays and the number of times that each player won should be output as well.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

Two players play this game. Players enter their choices via the keyboard. Each player enters `R` or `r` for Rock, `P` or `p` for Paper, or `S` or `s` for Scissors. While the first player enters a choice, the second player looks away. Once both entries are in, if the entries are valid, the program outputs the players' choices and declares the winner of the play. The game continues until one of the players decides to quit. After the game ends, the program outputs the total number of plays and the number of times that each player won. This discussion translates into the following algorithm:

1. Provide a brief explanation of the game and how it is played.
2. Ask the users if they want to play the game.
3. Get plays for both players.
4. If the plays are valid, output the plays and the winner.
5. Update the total game count and winner count.
6. Repeat Steps 2–5 while the users continue to play the game.
7. Output the number of plays and times that each player won.

To describe the objects ROCK, PAPER, and SCISSORS, we define the following enum type:

```
public enum RockPaperScissors
{
    ROCK ("Rock crushes scissors."),
    PAPER ("Paper covers rock."),
    SCISSORS ("Scissors cuts paper.");

    private String mgs;

    private RockPaperScissors()
    {
        mgs = "";
    }

    private RockPaperScissors(String str)
    {
        mgs = str;
    }

    public String getMessage()
    {
        return mgs;
    }
}
```

Variables (Method main)

It is clear that you need the following variables in the method main:

```
int gameCount;      //to count the number of
                    //games played
int winCount1;      //to count the number of
                    //games won by player 1
int winCount2;      //to count the number of
                    //games won by player 2
int gameWinner;
char response;      //to get the user's response
                    //to play the game
char selection1;
char selection2;

RockPaperScissors play1;  //player1's selection
RockPaperScissors play2;  //player2's selection
```

This program is divided into six methods, which the following sections describe in detail.

- `displayRules`: This method displays some brief information about the game and its rules.
- `validSelection`: This method checks whether a player's selection is valid. The only valid selections are `R`, `r`, `P`, `p`, `S`, and `s`.
- `retrievePlay`: This method uses the entered choice (`R`, `r`, `P`, `p`, `S`, or `s`) and returns the appropriate object.
- `gameResult`: This method outputs the players' choices and the winner of the game.
- `winningObject`: This method determines and returns the winning object.
- `displayResults`: After the game is over, this method displays the final results.

Method `displayRules`

This method has no parameters. It consists only of output statements to explain the game and rules of play. Essentially, this method's definition is:

```
public static void displayRules()
{
    System.out.println("Welcome to the game of Rock, "
                     + "Paper, and Scissors.");
    System.out.println("This is a game for two players. "
                     + "For each game, each player \n"
                     + "selects one of the "
                     + "objects: Rock, Paper or "
                     + "Scissors.");
    System.out.println("The rules for winning the "
                     + "game are: ");
    System.out.println("1. If both players select the "
                     + "same object, it is a tie.");
    System.out.println("2. Rock crushes Scissors: The "
                     + "player who selects Rock wins.");
    System.out.println("3. Paper covers Rock: The "
                     + "player who selects Paper wins.");
    System.out.println("4. Scissors cuts Paper: The "
                     + "player who selects Scissors "
                     + "wins.");
    System.out.println("Enter R or r to select Rock, "
                     + "P or p to select Paper, \n"
                     + "and S or s to select Scissors.");
}
```

Method `validSelection`

This method checks whether a player's selection is valid. Let's use a **`switch`** statement to check for the valid selection. The definition of this method is:

```
public static boolean validSelection(char selection)
{
    switch (selection)
    {
    case 'R':
    case 'r':
    case 'P':
    case 'p':
    case 'S':
    case 's':
        return true;

    default:
        return false;
    }
}
```

Method retrievePlay This method uses the entered choice (`R`, `r`, `P`, `p`, `S`, or `s`) and returns the appropriate object. The method has one parameter of the type `char`. It is a value-returning method and returns a reference to a `RockPaperScissors` object.

The definition of the method `retrievePlay` is:

```
public static RockPaperScissors retrievePlay
                                (char selection)
{
    RockPaperScissors obj = RockPaperScissors.ROCK;

    switch (selection)
    {
    case 'R':
    case 'r':
        obj = RockPaperScissors.ROCK;
        break;

    case 'P':
    case 'p':
        obj = RockPaperScissors.PAPER;
        break;

    case 'S':
    case 's':
        obj = RockPaperScissors.SCISSORS;
    }

    return obj;
}
```

Method `gameResult`

This method decides whether a game is a tie or which player is the winner. It outputs the players' selections and the winner of the game. This method has two parameters: player 1's choice and player 2's choice. It returns the number (1 or 2) of the winning player.

The definition of this method is:

```
public static int gameResult(RockPaperScissors play1,
                             RockPaperScissors play2)
{
    int winner = 0;

    RockPaperScissors winnerObject;

    if (play1 == play2)
    {
        winner = 0;
        System.out.println("Both players selected "
                         + play1
                         + ". This game is a tie.");
    }
    else
    {
        winnerObject = winningObject(play1, play2);

            //Output each player's choice
        System.out.println("Player 1 selected " + play1
                         + " and player 2 selected "
                         + play2 + ".");

            //Decide the winner
        if (play1 == winnerObject)
            winner = 1;
        else if (play2 == winnerObject)
            winner = 2;

            //Output winning object's message
        System.out.println(winnerObject.getMessage());

            //Output the winner
        System.out.println("Player " + winner
                         + " wins this play.");
    }

    return winner;
}
```

Method winningObject

To decide the winner of the game, you look at the players' selections and then at the rules of the game. For example, if one player chooses ROCK and another chooses PAPER, the player who chose PAPER wins. In other words, the winning object is PAPER. The method winningObject, given two objects, decides and returns the winning object. Clearly, this method has two parameters of the type RockPaperScissors, and the value returned by this method is also of the type RockPaperScissors. The definition of this method is:

```
public static RockPaperScissors winningObject
                      (RockPaperScissors play1,
                       RockPaperScissors play2)
{
    if ((play1 == RockPaperScissors.ROCK &&
         play2 == RockPaperScissors.SCISSORS)
       || (play2 == RockPaperScissors.ROCK &&
           play1 == RockPaperScissors.SCISSORS))
        return RockPaperScissors.ROCK;
    else if ((play1 == RockPaperScissors.ROCK &&
              play2 == RockPaperScissors.PAPER)
             || (play2 == RockPaperScissors.ROCK &&
                 play1 == RockPaperScissors.PAPER))
        return RockPaperScissors.PAPER;
    else
        return RockPaperScissors.SCISSORS;
}
```

Method displayResults

After the game is over, this method outputs the final results—that is, the total number of plays and the number of plays won by each player. The total number of plays is stored in the variable gameCount, the number of plays by player 1 is stored in the variable winCount1, and the number of plays won by player 2 is stored in the variable winCount2. This method has three parameters corresponding to these three variables. Essentially, the definition of this method is as follows:

```
public static void displayResults(int gCount, int wCount1,
                                  int wCount2)
{
    System.out.println("The total number of plays: "
                     + gCount);
    System.out.println("The number of plays won by "
                     + "player 1: " + wCount1);
    System.out.println("The number of plays won by "
                     + "player 2: " + wCount2);
}
```

We are now ready to write the algorithm for the method main.

Main Algorithm

```
1. Declare the variables.
2. Initialize the variables.
3. Display the rules.
4. Prompt the users to play the game.
5. Get the users' responses to play the game.
6. while (response is yes)
   {
   a. Prompt player 1 to make a selection.
   b. Get the play for player 1.
   c. Prompt player 2 to make a selection.
   d. Get the play for player 2.
   e. If both the plays are legal
      {
        i. Retrieve both plays.
       ii. Increment the total game count.
      iii. Declare the winner of the game.
       iv. Increment the winner's game win count by 1.
      }
   f. Prompt the users to determine whether they want to play again.
   g. Get the players' responses.
   }
7. Output the game results.
```

PROGRAM LISTING

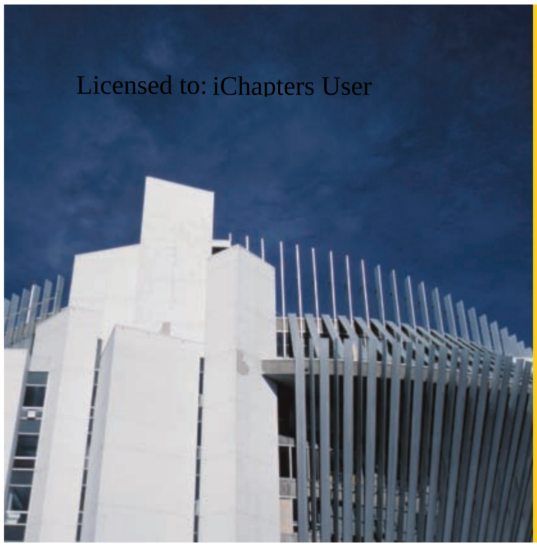
```
import java.util.*;

public class GameRockPaperScissors
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
              //Step 1
        int gameCount; //to count the number of
                       //games played
        int winCount1; //to count the number of
                       //games won by player 1
        int winCount2; //to count the number of
                       //games won by player 2
```

```
int gameWinner;
char response;  //to get the user's response
                //to play the game
char selection1;
char selection2;

RockPaperScissors play1;  //player1's selection
RockPaperScissors play2;  //player2's selection

    //Initialize the variables; Step 2
gameCount = 0;
winCount1 = 0;
winCount2 = 0;

displayRules();                              //Step 3

System.out.print("Enter Y/y to play "
               + "the game: ");              //Step 4
response = console.nextLine().charAt(0);     //Step 5
System.out.println();

while (response == 'Y' || response == 'y')   //Step 6
{
    System.out.print("Player 1 enter "
                   + "your choice: ");       //Step 6a
    selection1 =
           console.nextLine().charAt(0);     //Step 6b
    System.out.println();

    System.out.print("Player 2 enter "
                   + "your choice: ");       //Step 6c
    selection2 =
            console.nextLine().charAt(0);    //Step 6d
    System.out.println();

        //Step 6e
    if (validSelection(selection1) &&
        validSelection(selection2))
    {
        play1 = retrievePlay(selection1);
        play2 = retrievePlay(selection2);
        gameCount++;
        gameWinner = gameResult(play1, play2);

        if (gameWinner == 1)
            winCount1++;
        else if (gameWinner == 2)
            winCount2++;
    }//end if
```

```
                System.out.print("Enter Y/y to play "
                               + "the game: ");          //Step 6f
                response = console.nextLine().charAt(0); //Step 6g
                System.out.println();
            }//end while

            displayResults(gameCount, winCount1,
                           winCount2);                   //Step 7

        }//end main

        //Place the definitions of the methods displayRules,
        //validSelection, retrievePlay, winningObject,
        //gameResult, and displayResults here.
}
```

Sample Run: (In this sample run, the user input is shaded.)

```
Welcome to the game of Rock, Paper, and Scissors.
This is a game for two players. For each game, each player
selects one of the objects: Rock, Paper or Scissors.
The rules for winning the game are:
1. If both players select the same object, it is a tie.
2. Rock crushes Scissors: The player who selects Rock wins.
3. Paper covers Rock: The player who selects Paper wins.
4. Scissors cuts Paper: The player who selects Scissors wins.
Enter R or r to select Rock, P or p to select Paper,
and S or s to select Scissors.
Enter Y/y to play the game: y

Player 1 enter your choice: R

Player 2 enter your choice: S
Player 1 selected ROCK and player 2 selected SCISSORS.
Rock crushes scissors.
Player 1 wins this play.
Enter Y/y to play the game: Y

Player 1 enter your choice: S

Player 2 enter your choice: P
Player 1 selected SCISSORS and player 2 selected PAPER.
Scissors cuts paper.
Player 1 wins this play.
Enter Y/y to play the game: Y

Player 1 enter your choice: R
Player 2 enter your choice: P
Player 1 selected ROCK and player 2 selected PAPER.
Paper covers rock.
Player 2 wins this play.
Enter Y/y to play the game: n
The total number of plays: 3
The number of plays won by player 1: 2
The number of plays won by player 2: 1
```



Index

Note: Page numbers in **boldface type** indicate pages where key terms are defined.

Special Characters

\ (backslash), 68, 70
< (left angle bracket), 180
> (right angle bracket), 180
! (exclamation point), 180, 183–184
\" (double quotation escape sequence), 70
& (ampersand), 183, 184, 202
\' single quotation escape sequence), 70
+ (plus sign), 46–48, 64–66
- (minus sign), 64–65, 66
\b (backspace escape sequence), 70
= (equal sign), 51, 180, 181
? (question mark), 204
. (dot), 120–121, 476
; (semicolon), 81
\n (newline escape sequence), **68,** 70
(|) pipe, 183, 185, 202
\r (return escape sequence), 70
\t (tab escape sequence), 70

A

abacus, 2
abs method, 385
abstract classes, **675,** 675–681
abstract data types (ADTs), 517–518, **518**
abstract methods, **674,** 674–681
accessing array elements, 555–557
  two-dimensional arrays, 591–594
accessing class members, 475–476
accessor methods, **507,** 507–510
action events, **348**
action listeners, **348**
  registering, **349**
action statements, **190**
actual parameter(s), **393**
actual parameter lists, syntax, 394–395, 408
address, memory cells, **5**
addWindowListener method, 762–763
ADTs (abstract data types), 517–518, **518**
Advanced Research Project Agency (ARPA), 13
aggregation, **640, 684,** 684–709
Aiken, Howard, 3
algorithms, **14,** 15, 16–19
  recursive, **875,** 877
  short-circuit evaluation, 201–202
American Standard Code for Information Interchange (ASCII), **7**
ampersand (&), and operator, 183, 184, 202
analog signals, **6**
and operator (&&), 183, 184, 202
Apple computers, 3
applets, 28, **784, 787,** 787–811
  Color class, 794–799
  Component class, 785–786
  Container class, 786–787
  converting an application program to, 808–811
  Font class, 791–794
  Graphics class, 800–808
  JApplet class, 787–788
application programs, **6,** 26–31
  comments, 29–30
  converting to an applet, 808–811
  creating, 72–76
  flow of execution, 404–406
  identifiers, 31
  keywords, **30**
  processing, 10–12
  special symbols, 30
arguments, methods, **385**
ArithmeticException class, 737
arithmetic expressions, **36,** 40–43
  character arithmetic, **40**
  floating-point (decimal), **40,** 40–41
  integral, **40**
  mixed, **41,** 41–43
  type conversion (casting), 43–48

arithmetic operators, **36,** 36–40
  associativity, **40**
  order of precedence, 39–40
ARPA (Advanced Research Project Agency), 13
ARPANET, 13
array(s), 551–623, **553**
  assignment operator, 565–566
  base address, 564
  declaring. *See* declaring arrays
  elements. *See* accessing array elements; array elements
  indexes, **555**
  `length` instance variable, 558–559
  multidimensional, 603–615
  need for, 552–553
  of objects, 574–581
  one-dimensional. *See* one-dimensional arrays
  in and out of bounds indexes, 564
  as parameters to methods, 567–571
  ragged, **593**
  relational operators, 566–567
  searching for a specific item, 572–574
  size, 559
  specifying size during program execution, 557
  two-dimensional. *See* two-dimensional arrays
  variable length parameter list, 581–589
  `Vector` class, 616–621
array elements, **553**
  accessing. *See* accessing array elements
  largest, finding, 561–562
`ArrayIndexOutOfBounds Exception` class, 737
array subscripting operator, **555**
ASCII (American Standard Code for Information Interchange), **7**
assemblers, **9**
assembly language, **8,** 8–9
assignment operator (=), **51, 55,** 181
  arrays, 565–566
  associativity, **55**
  classes, 476–478
assignment statements
  compound, 85–86
  putting data into variables, 51–55
  simple, 85–86
associativity of operators, **186**
  arithmetic operators, **40**
  assignment operator, **55**
Augusta, Ada, 2
autoboxing, **373**
auto-unboxing, **373**
avoiding bugs
  by avoiding partially understood concepts and techniques, 204–208, 215–217
  by avoiding patches, 295–298
  with consistent, proper formatting and code walk-through, 84–85
  with one-piece-at-a-time coding, 442

B

Babbage, Charles, 2
  backslash (\),escape sequences, 68, 70
backspace escape sequence (\b) 70
base 2, **7**
base 10, **7**
base address, arrays, 564
base case, recursion, **874**
base classes. *See* superclasses
`BButton` class, methods, 347
binary code (binary numbers), **6**
  conversion from decimal to (programming example), 889–892
binary digits (bits), **6**
binary operators, **36**
binary searches, **917,** 917–933
  performance, 920
binary system, 7
binary units, 7
binding, late (dynamic or run-time), **664,** 665
blanks, 81
blocks of statements, **197**
body, methods, **74**
Boolean data type, **32**
`boolean` data type, 189
`boolean endsWith` method, 124
`boolean equals` method, 123
Boolean expressions, **34,** 189
  as part of `if` statement, using strings in, 227–228
Boolean operators, **183,** 183–185, 202
`boolean regionMatches` method, 124
`boolean startsWith` method, 124
`BorderLayout` manager, 843–844
bottom-up design, **19**
branches, **178**
`break` statement, 293–295
browsers, 13
bugs. *See* avoiding bugs; debugging
built-in operations, classes, 476
byte(s), **6,** 7
bytecode, **9**

C

cable company billing (programming example), 217–223
calculating machines, 2
calculator (programming example), 766–775
candy machine (programming example), 519–536
cascaded method calls, 514–517
case sensitivity, 31
cast operator, **43,** 43–48
`catch` blocks, 728–733
  rethrowing exceptions caught by, 750–751
CDs (certificates of deposit), **676**
`ceil` method, 386

central processing unit (CPU), **4,** 4–5
certificates of deposit (CDs), **676**
change, making (programming example), 91–94
character(s), single, reading, 61–63
character arithmetic, **40**
`Character` class, 387–388
  predefined methods, 387–388
character literals (character constants or characters), **36**
character string, **45**
`char charAt` method, 122
check boxes, 816–823
checked exceptions, **741,** 741–742
checking accounts, **676**
`Circle` class, 467
`class` container, methods, 787
classes, **20, 71,** 466–494, **468**
  abstract, **675,** 675–681
  accessing members, 475–476
  assignment operator, 476–478
  built-in operations, 476
  clients, **490**
  constructors, **471,** 471–472
  designing and documenting, 510–512
  implementing, 370–377
  inner, **517**
  instance methods, **471**
  instances, 475
  members (fields), **468.** *See also* class members
  methods. *See* method(s); method(s) (listed by name)
  object instantiation, 464–465
  objects, 466
  operations, 370–377
  protected members, 657–669
  scope, 478
  syntax for defining, 468
  `toString` method, 494–500
  UML class diagrams, 472–473, **473**
  variable declaration, 473–474
classes (listed by name)
  `ArithmeticException` class, 737
  `ArrayIndexOutOfBounds Exception` class, 737
  `Character` class, 387–388
  `Circle` class, 467
  `Clock` class, 479–490, 501
  `Color` class, 794–799
  `Component` class, 785–787
  `Container` class, 786–787
  `Double` class, 376–377, 740
  `Exception` class, 736–741, 746–749, 758–760
  `FileNotFoundException` class, 737
  `Font` class, 791–794
  `Graphics` class, 800–808
  `IllegalArgument Exception` class, 737
  `IndexOutOfBounds Exception` class, 737
  `InputMismatch Exception,` 739
  `Integer` class, 370–376, 740
  `JApplet` class, 787–788
  `JButton` class, 347–348
  `JCheckbox` class, 817–823
  `JComboBox` class, 828–833
  `JFrame` class, 332–338
  `JLabel` class, 339–343
  `JList` class, 833–839
  `JOptionPane` class, 139–145
  `JRadioButton` class, 823–828
  `JTextArea` class, 811–816
  `JTextBox` class, 816–823
  `JTextField` class, 343–346
  `Math` class, 385–387
  `NullPointerException` class, 737
  `NumberFormatException` class, 738
  `Object` class, 661–663
  `StringBuffer` class, 418–421
  `String` class, 121–129, 146–149, 466–467, 741
  `StringIndexOutOf Bounds Exception` class, 741
  `Vector` class, 616–621
classifying numbers (programming example), 285–288
class libraries, **118, 385**
class members, **468,** 468–469
  accessing, 475–476
  `private,` 468–469, 508, 642, 657
  protected, 657–669
  `public,` 468–469, 508
  static members, 501–507
clients, classes, **490**
`Clock` class, 469–490
  accessing members, 475–476
  constructors, 479, 486–487
  definition, 487–490
  methods, 479–486
  static members, 501
  UML diagram, 472–473
  variable declaration and object instantiation, 473–475
COBOL, 3
code detection (programming example), 605–609
code walk-through, avoiding bugs using, 84–85
collating sequences, **34**
`Color` class, 794–799
`Color` method, 794, 795
column processing, two-dimensional arrays, **595**
combo boxes, **828,** 828–833
comments, 29–30
  multiple-line, **30**
  single-line, **29,** 29–30
`CompareTo` method, 223–230
comparing strings, 223–230
compilers, **9,** 9–10
`Component` class, 785–786
  constructors and methods, 785–787

composition, **640, 684,** 684–709
compound assignment statements, 85–86
compound statements, **197**
computer programs, **26**
computer systems, elements, 4–6
concatenation operator (+), strings, 46–48
condition(s), **190, 251**
conditional expressions, **204**
conditional operator (?), **204**
constants
  allocating memory with named constants and variables, 48–51
  character (character literals or characters), **36**
  floating-point (floating-point literals or floating-point numbers), **36**
  integer (integer literals or integers), **36**
  named, **48,** 48–51
  string (string literals), **45**
constructors, **334, 471,** 471–472
  `Clock` class, 479, 486–487
  copy, **500,** 500–501
  default, **471,** 471–472
  superclass and subclass, 648–657
`Container` class, 786–787
  methods, 338–339
content pane, **332**
  getting access, 338–354
`continue` statement, 293–295
control expressions, **278**
control structures, 177–179
  repetition. *See* repetition
  selection. *See* selection
conversions, 131–134
converting length (programming example), 86–90
copy constructors, **500,** 500–501
copying
  deep, **478**
  shallow, **477,** 477–478
`cos` method, 387
counted `for` loops, **279**
counter-controlled `while` loops, **255,** 255–257
CPU (central processing unit), **4,** 4–5

D

data, putting into variables, 51–55
data comparison (programming example), 429–439
data types, **32**
  abstract, 517–518, **518**
  primitive. *See* primitive data types
debugging
  designing and documenting classes, 510–512
  drivers and stubs, 440–442
  logic errors, 163–165
  syntax errors, 77–80
decimal expressions, **40,** 40–41
decimal system, **7**
decimal to binary conversion (programming example), 889–892
declaration statements, **74**
declaring arrays, 553
  as formal parameters to methods, 564–565
  initialization during declaration, 558
  initializing arrays during declaration, 594–595
declaring variables, 55–56, 473–474
decrement operator (--), **64,** 64–65, 66
deep copying, **478**
default constructors, **471,** 471–472
derived classes. *See* subclasses
dialog boxes, input/output, 139–145
digital signals, **6**
directly recursive methods, **876,** 876–877
Discrete Mathematics: Theory and Applications (Malik and Sen), 298
`do. . .while` loops, 288–293
documenting classes, 510–512
dot (.), member access operator, **120,** 120–121, 476
`Double` class, 376–377, 740
double precision, **35**
double quotation escape sequence (\"), 70
driver programs, **440,** 440–441
drop-down lists, 828–833
dynamic binding, **664**

E

early binding, 665
EBs (exabytes), 7
election results (programming example), 921–933
Electronic Numerical Integrator and Calculator (ENIAC), 3
empty string, **45**
encapsulation, **363**
End Of File (EOF)-controlled `while` loops, **266,** 266–271
ENIAC (Electronic Numerical Integrator and Calculator), 3
EOF (End Of File)-controlled `while` loops, **266,** 266–271
equality operator (==), 180, **181**
equal sign (=), 180, 181
  assignment operator (=), **51,** 181
  equality operator (==), **181**
  greater than or equal to operator (>=), 180
  less than or equal to operator (>=), 180
  not equal to operator (!=), 180
`equals` method, 228, 479, 482–483
equal to operator (==), 180, 181
errors. *See* avoiding bugs; debugging; exception handling
escape character **68,** 70
event(s), **332**
  action, **348**
  handling, 348–354
event handling, 760–775

exabytes (EBs), 7
exception(s), throwing, 153
Exception class, 736–741
  creating, 758–760
  instanceof operator, 746–749
exception handling, 723–760
  checked and unchecked exceptions, 741–742
  creating exception classes, 758–760
  Exception class, 736–741, 746–749
  FileNotFound Exception, 152–153
  by fixing the error and continuing, 756–757
  instanceof operator, 746–749
  Java exception hierarchy, 733–736
  Java's mechanism, 727–728
  by logging the error and continuing, 757
  printStackTrace method, 753–755
  within a program, 724–733
  rethrowing and throwing an exception, 749–753
  by terminating the program, 755
  try/catch/finally blocks, 728–733
exclamation point (!), 180, 183–184
  not equal to operator (!=), 180
  not operator (!), 183–184
executable statements, **74**
exp method, 386
expressions
  arithmetic. *See* arithmetic expressions
  Boolean (logical), **34,** 189, 190
  conditional, **204**
  control, **40,** 40–41, **278**
  floating-point (decimal), **40,** 40–41
  integral, **40**
  mixed, **41,** 41–43

F

Fibonacci number, **273**
  programming example, 273–278
Fibonacci sequence, **273**
fields, **364**
  classes, **468**
  text, **140,** 811–816
file(s), **149**
  input/output, 149–154
  storing output to, 152–153
FileNotFoundException class, 737
FileNotFoundException exception, 152–153
finalizers, **507**
final keyword, 669
finally blocks, 728–733
  order, 729–730
  using in a program, 730–733
flag-controlled while loops, **263,** 263–266
flag variables, **263**
floating-point data types, **32,** 34–35
floating-point expressions, **40,** 40–41
floating-point literals (floating-point constants or floating-point numbers), **36**
  comparing for equality, 202–203
floating-point notation, **35**
floor method, 386
FlowLayout manager, 840–842
Font class, 791–794
foreach loops, Vector class, 620–621
for loops, 278–284
  counted (indexed), **279**
formal parameter(s), **392,** 392–393
  declaring arrays as, 564–565
formal parameter lists
  different, 427
  syntax, 394, 407
format specifiers, **130,** 130–131, 135–138
formatting for avoiding bugs, 84
FORTRAN, 3
fractals, **892**

G

garbage collection, **117**
GBs (gigabytes), 7
general case, recursion, **874**
getContentPane method, 338
getCopy method, 478, 479, 484–486
getHours method, 479, 481
getMinutes method, 479, 481
getSeconds method, 479, 481
gigabytes (GBs), 7
grade report (programming example), 690–709
graphical user interfaces (GUIs), 328–362, **330,** 811–865
  addWindowListener method, 762–763
  creating windows, 332–338
  events generated by components, 764–765
  getting access to content pane, 338–354
  interface MouseListener, 763–764
  Java kiosk (programming example), 857–865
  JComboBox class, 828–833
  JList class, 833–839
  JRadioButton class, 823–828
  JTextArea class, 811–816
  JTextBox class, 816–823
  key events, 848–850
  layout managers, 839–844
  menus, 844–847
  mouse events, 850–857
Graphics class, 800–808
greater than operator (>), 180
greater than or equal to operator (>=), 180
GUIs. *See* graphical user interfaces (GUIs)

H

hardware, 4–5
hasNext method, exceptions thrown by, 739

headings, methods, **74**
high-level languages, **9**
history of computers, 2–3
Hollereith, Herman, 3
Hypertext Markup Language (HTML), 13

I

IBM, 3
IBM PC, 3
IDE (integrated development environment), **11,** 12
identifiers, **31**
  declaring, 51
  local, **422**
  scope, **422,** 422–427
`if. . .else` statements, 190–204. *See also* selection
  choosing between `switch` structures and, 215
  comparing with a series of `if` statements, 200–201
`if` statements, 190–204. *See also* selection
  series of, comparing `if. . .else` statements with, 200–201
  using strings in Boolean expressions as part of, 227–228
`IllegalArgumentException` class, 737
immutability, **372**
implicit type coercion, **43**
in bounds, arrays, 564
`incrementHours` method, 479, 481
`incrementMinutes` method, 479, 481
increment operator (++), **64,** 64–66
`incrementSeconds` method, 479, 482
indenting programs, 208
index(es), **122**
  arrays, **555**
  in and out of bounds, 564
indexed `for` loops, **279**
`IndexOutOfBoundsException` class, 737
indirectly recursive methods, **876,** 876–877
infinite loops, **252**
infinite recursion, 877
information hiding, **518**
inheritance, **334, 640,** 640–663
  constructors of the superclass and subclass, 648–657
  Java stream classes, 663–664
  multiple, **642**
  `Object` class, 661–663
  protected access vs. package access, 660–661
  protected members of a class, 657–660
  single, **642**
  using superclass methods in a subclass, 642–648
initializer lists, **558**
initializing arrays during declaration, 558
initializing variables, **51,** 55–56, 60–61
initial values, **558**
inner classes, **517**
input, 48–63
  allocating memory with named constants and variables, 48–51
  declaring and initializing variables, 55–56
  input (read) statements, 56–61
  putting data into variables, 51–55
  reading a single character, 61–63
input devices, **5**
`InputMismatchException,` 739
input/output (I/O), files, 149–154
input/output (I/O) classes, 129–149
  dialog boxes, 139–145
input statement, 56–61
insertion sort, 913–917
instance(s), classes, 475
instance methods, **471**
`instanceof` operator, 670–674
instance variables, **469,** 469–470
  `length` instance variable, 558–559, 692–594
instantiation, objects, **116,** 474–475
`int compareTo` method, 123, 371
`Integer` class, 370–376, 740
  members, 371–372
integer literals (integer constants or integers), **36**
integral data types, **32,** 32–34
integral expressions, **40**
integrated development environment (IDE), **11,** 12
interface(s), 681–682. *See also* graphical user interfaces (GUIs); user interfaces
  polymorphism via, 682–684
`interface MouseListener,` 763–764
interpreters, **11,** 11–12
`int indexOf` method, 122, 123
`int length` method, 123
I/O. *See* input/output (I/O) *entries*
`isDigit` method, 387
`isLetter` method, 387
`isLowerCase` method, 388
`isUpperCase` method, 388
item comparisons, **908**
iteration, recursion compared, 888–896
iterative control structures, **888**

J

Jacquard, Joseph, 2
`JApplet` class, 787–788
  members, 787–788
Java kiosk (programming example), 857–865
Java stream classes, 663–664
Java Virtual Machine (JVM), **10**
`JButton` class, 347–348
`JCheckbox` class, 817–823
`JComboBox` class, 828–833
`JFrame` class, 332–338
`JLabel` class, 339–343
`JList` class, 833–839

Jobs, Steven, 3
JOptionPane class, 139–145
JRadioButton class, 823–828
JTextArea class, 811–816
JTextBox class, 816–823
JTextField class, 343–346
JVM (Java Virtual Machine), **10**

K

KBs (kilobytes), **6,** 7
key comparisons, **908**
key component, events generated by, 765–766
key events, 848–850
keywords, **30,** 468–469
kilobytes (KBs), **6,** 7

L

largest element in the array problem, 878–881
largest number (programming example), 405–406
late binding, **664,** 665
layout managers, 839–844
  BorderLayout, 843–844
  FlowLayout, 840–842
left angle bracket (<), 180
  less than operator (<), 180
  less than or equal to operator (<=), 180
length, converting (programming example), 86–90
length instance variable, 558–559
  two-dimensional arrays, 592–594
length of string, **45**
less than operator (<), 180
less than or equal to operator (<=), 180
Liebniz, Gottfried von, 2
linear searches, **573**
lists, **833,** 833–839, **908,** 908–933
  binary search, **917,** 917–933
  insertion sort, 913–917
  searching, 908–909
  selection sort, **909,** 909–913
literals, 36.
  See also constants
loaders, **11**
local identifiers, **422**
logical expressions, **34,** 189, 190
  short-circuit evaluation, 201–202
logical operators, **183,** 183–185, 202
logic errors, debugging, 163–165
log method, 386
log10 method, 386
loop(s), **178.**
  See also repetition
loop conditions, **251, 278, 289**
loop control variables, **253**

M

machine independence, **10**
machine language, **6,** 6–8, 9–10
main memory, **4,** 4–5
main method, 72–73, 74
makeCopy method, 477–478, 479, 483–484
making change (programming example), 91–94
Malik, D. S., 298
Mandelbrot, Benoit, 892
Mark I, 3
Math class, methods and named constants, 385–387
max method, 386
megabytes (MBs), 7
member(s), classes. *See* class members
member access operator (.), **120,** 120–121, 476
memory
  allocating with named constants and variables, 48–51
  garbage collection, **117**
  main, **4,** 4–5
memory cells, **5**
menus, 844–847
method(s), **20, 71, 118,** 479
  abstract, **674,** 674–681
  accessor, **507,** 507–510
  body, **74**
  cascaded method calls, 514–517
  Clock class, 479–486
  directly recursive, **876,** 876–877
  final, 669
  finalizers, **507**
  headings, **74**
  indirectly recursive, **876,** 876–877
  instance, **471**
  mutator, **507,** 507–510
  nonstatic, **389**
  parameters (arguments), **385**
  passing two-dimensional arrays as parameters to, 599–603
  predefined, **72,** 118–129, **384,** 384–391
  recursive, **875,** 877
  signature of, **427**
  standard, **72**
  String class, 122–124
  stubs, **441,** 441–442
  superclasses, using in subclasses, 642–648
  tail recursive, **877**
  user-defined. *See* user-defined methods
method(s) (listed by name)
  abs method, 385
  addWindowListener method, 762–763
  BButton class, methods, 347
  boolean endsWith method, 124
  boolean equals method, 123
  boolean regionMatches method, 124
  boolean startsWith method, 124
  ceil method, 386
  char charAt method, 122
  CompareTo method, 223–230
  cos method, 387
  equals method, 228, 479, 482–483
  exp method, 386
  floor method, 386

method(s) (listed by name) (*Continued*)
getContentPane method, 338
getCopy method, 478, 479, 484–486
getHours method, 479, 481
getMinutes method, 479, 481
getSeconds method, 479, 481
hasNext method, 739
incrementHours method, 479, 481
incrementMinutes method, 479, 481
incrementSeconds method, 479, 482
int compareTo method, 123, 371
int indexOf method, 122, 123
int length method, 123
isDigit method, 387
isLetter method, 387
isLowerCase method, 388
isUpperCase method, 388
log method, 386
log10 method, 386
main method, 72–73, 74
makeCopy method, 477–478, 479, 483–484
max method, 386
min method, 386
nextDouble method, 738
nextInt method, 738
nextLine method, 739
next method, 739
parseInt (String str), 740
pow method, 386
printf method, 129–139
printStackTrace method, 753–755
printTime method, 479, 481
protected Object clone method, 662
protected void finalize method, 662
public boolean contains method, 616
public boolean equals method, 371
public boolean isEmpty method, 617
public Component add method, 787
public double doubleValue method, 371
public Exception method, 737
public int indexOf method, 616, 617
public int intValue method, 371
public int size method, 617
public JButton method, 347
public JFrame method, 333
public Object elementAt method, 616
public Object method, 662
public static Integer valueOf method, 372
public static int parseInt method, 372
public static String toString method, 372
public string getMessage method, 734
public String getText method, 344, 347
public String toString method, 372, 617, 662, 734
public void addActionListener method, 347
public void addElement method, 616
public void add method, 339
public void addWindowListener method, 334
public void insertElementAt method, 616
public void printStackTrace method, 734
public void removeAllElements method, 617
public void removeElementAt method, 617
public void setDefaultClose Operation method, 333
public void SetSize method, 333
public void setText method, 347
public void SetTitle method, 333
public void setVisible method, 333
public void update method, 787
public void validate method, 787
round method, 386
setTime method, 479–480
showInputDialog method, 139–140
showMessageDialog method, 139, 140–145
sin method, 387
sqrt method, 387
String concat method, 123
String methods, 121–129, 146–149
String replace method, 123
String substring method, 124
String toLowerCase method, 124
String toUpperCase method, 124
tan method, 387
toLowerCase method, 388
toString method, 494–500
toUpperCase method, 388
valueOf, 740
method calls, syntax, 407

method overloading, **427,** 427–439
method type, **385**
min method, 386
minus sign (-), decrement operator, 64–65, 66
mixed expressions, **41,** 41–43
mnemonics, **8**
mod (modulus or remainder), **36**
modular programming, **19**
mouse component, events generated by, 765–766
mouse events, 850–857
movie ticket sale and donation (programming example), 154–159
multidimensional arrays, **603,** 603–615
multiple inheritance, **642**
multiple-line comments, **30**
multiple selections, 198–200
mutator methods, **507,** 507–510

N

named constants, **48,** 48–51
  Math class, 385–386
n-dimensional arrays, **603,** 603–615
nested if. . .else statements, 198–200
nesting, **284,** 284–288
Neumann, John von, 3
newline escape sequence (\n), **68,** 70
new operator, 229–230
nextDouble method, exceptions thrown by, 738
nextInt method, exceptions thrown by, 738
nextLine method, exceptions thrown by, 739
next method, exceptions thrown by, 739
nonstatic methods, **389**
not equal to operator (!=), 180
not operator (!), 183–184
NullPointerException class, 737
NullPointerException exceptions, 741
null string, **45**

number(s)
  binary. *See* binary code (binary numbers)
  classifying (programming example), 285–288
  Fibonacci. *See* Fibonacci number
  floating-point. *See* floating-point literals (floating-point constants or floating-point numbers)
  largest (programming example), 405–406
NumberFormatException class, 738
NumberFormatException exceptions, 740
numeric strings, **138**
  parsing, **138,** 138–139

O

object(s), **19, 20, 116**
  arrays, 574–581
  immutability, **372**
  instantiation, **116,** 474–475
Object class, inheritance, 661–663
object-oriented design (OOD), **19,** 363–370
  simplified methodology, 364–370
object-oriented programming (OOP), 19–20, **20**
one-dimensional arrays, **553**
  processing, 559–563
one-way selections, 190–193
OOD. *See* object-oriented design (OOD)
OOP (object-oriented programming), 19–20, **20**
operands, **36**
operating systems, **6**
operators
  arithmetic. *See* arithmetic operators
  assignment. *See* assignment operator (=)
  associativity of. *See* associativity of operators
  binary, **36**
  Boolean (logical), **183,** 183–185, 202
  order of precedence, 39–40, 185–189
  relational. *See* relational operators
  ternary, **204**
  unary, **36**
order of precedence, 39–40, 185–189
or operator (||), 183, 185, 202
out of bounds, arrays, 564
output
  formatting using String method format, 146–149
  storing to a file, 152–153
output devices, **5**
output statements, **66,** 66–71
overloading a method name, **427,** 427–439
overriding, **643**

P

package(s), **71,** 468
package access, inheritance, 660–661
palindromes, **403**
parameters, **119**
  actual, **393**
  arrays as, 567–571
  formal, **392,** 392–393
  methods, **385**
  passing two-dimensional arrays to methods as, 599–603
  primitive data type variables as, 411–414
  reference variables as, 414–421
  showMessageDialog method, 141
  wrapper classes as, 421
parseInt, 740
parsing numeric strings, **138,** 138–139
Pascal, Blaise, 2
Pascaline, 2
patches, avoiding bugs by avoiding, 295–298
petabytes (PBs), 7
pipe (|), or operator, 183, 185, 202
pixels, **335**

plus sign (+)
  concatenation operator (+), 46–48
  increment operator (++), **64,** 64–66
polymorphic reference variables, **665**
polymorphism, 664–674, **665**
  instanceof operator, 670–674
  via interfaces, 682–684
  polymorphic reference variables, **665**
pow method, 386
precision, **35**
predefined classes, **118,** 118–129
predefined methods, **72,** 118–129, **384,** 384–391
  using in a program, 388–391
primitive data types
  autoboxing, **373,** 373–376
  auto-unboxing, **373,** 373–376
  Boolean, **32**
  floating-point, **32,** 34–35
  implicit type coercion, **43**
  integral, **32,** 32–34
  relational operators, 181–183
  variables as parameters, 411–414
  Vector class, 620
  wrapper classes. *See* wrapper classes
printf method, 129–139
println statement, debugging using, 163–165
printStackTrace method, 753–755
print statement, debugging using, 163–165
printTime method, 479, 481
private class members, 468–469, 508, 642, 657
problem analysis–coding–execution cycle, 13–19
problem-solving process, 13–19
problem solving using recursion, 878–888
processing
  application programs, 10–12
  column, two-dimensional arrays, **595**
  one-dimensional arrays, 559–563
  row, two-dimensional arrays, **595**
  text (programming example), 609–615
  two-dimensional arrays, 595–599
program(s)
  applets. *See* applets
  application. *See* application programs
  computer, **26**
  driver, **440,** 440–441
  source, **10,** 10–11
programming, **26**
programming examples
  cable company billing, 217–223
  calculator, 766–775
  candy machine, 519–536
  classifying numbers, 285–288
  code detection, 605–609
  converting length, 86–90
  data comparison, 429–439
  decimal to binary conversion, 889–892
  election results, 921–933
  Fibonacci number, 273–278
  grade report, 690–709
  Java kiosk, 857–865
  largest number, 405–406
  making change, 91–94
  movie ticket sale and donation, 154–159
  Sierpinski gasket, 892–896
  student grade, 160–162
  temperature conversion, 355–362
  text processing, 609–615
programming languages, **29**
  creating Java programs, 12
  evolution, 8–10
  semantic rules, **29**
  syntax rules, **29.** *See also* syntax
prompt lines, **81,** 81–82
protected access, inheritance, 660–661
protected Component, 785
protected Graphics, 800
protected members, classes, 657–669
protected Object clone method, 662
protected String paramString, 788
protected void finalize method, 662
pseudocode, **19**
public abstract boolean drawImage, 800
public abstract Color getColor, 801
public abstract Font getFont, 802
public abstract void drawArc, 800
public abstract void drawLine, 800
public abstract void drawOval, 800
public abstract void drawPolygon, 800
public abstract void drawRect, 800
public abstract void drawRoundRect, 801
public abstract void drawString, 801
public abstract void fillOval, 801
public abstract void fillPolygon, 801
public abstract void fillRect, 801
public abstract void fillRoundRect, 801
public abstract void fill3DRect, 801
public abstract void Font setFont, 802
public abstract void setColor, 801
public boolean contains method, 616
public boolean equals, 795
public boolean equals method, 371, 662
public boolean isEmpty method, 617

public boolean isSelected
JCheckbox class, 818
JRadioButton class, 824
public boolean isVisible, 786
public boolean setSelected, 818, 824
public class members, 468–469, 508
public Color brighter, 795
public Color darker, 795
public Color getBackground, 785
public Color getForeground, 785
public Component add, 787
public Container getContentPane, 788
public double doubleValue method, 371
public Exception method, 737
public Font, 791
public font getFont, 786
public InputMismatch Exception, 739
public InputMismatch Exception constructor, 739
public integer constructor, 371
public int getBlue, 795
public int getGreen, 795
public int getRed, 795
public int getRGB, 795
public int getSelectedIndex, 834
public int indexOf method, 616, 617
public int intValue method, 371
public int size method, 617
public JButton method, 347
public JCheckBox, 817, 818
public JComboBox, 829
public JFrame method, 333
public JLabel, 340
public JList, 833, 834
public JMenuBar getJMenuBar, 788
public JRadioButton, 823, 824
public JTextArea, 812
public JTextField, 343, 344
public Object elementAt method, 616
public Object method, 662
public static Integer valueOf method, 372
public static int parseInt method, 372
public static String toString method, 372
public String getFamily, 791
public String getFontName, 791
public string getMessage method, 734
public String getText method, 344, 347, 812
public String toString
Color class, 795
Component class, 786
public String toString method, 372, 617, 662, 734
public Throwable, 734
public Throwable constructor, 734
public URL getCodeBase, 788
public URL getDocumentBase, 788
public void addActionListener method, 344, 347
public void add ComponentListener, 785
public void addElement method, 616
public void addFocusListener, 785
public void addKeyListener, 785
public void addList SelectionListener, 834
public void add method, 339
public void addMouseListener, 785
public void addMouse MotionListener, 785
public void addWindowListener, 334
public void append, 812
public void destroy, 788
public void draw3DRect, 800
public void init, 787
public void insertElementAt method, 616
public void paint, 786
public void printStackTrace method, 734
public void removeAllElements method, 617
public void remove ComponentListener, 785
public void removeElementAt method, 617
public void removeKeyListener, 785
public void removeMouseListener, 785
public void remove MouseMotionListener, 785
public void repaint, 786
public void setBackground, 785
public void setColumns, 812
public void setDefault CloseOperation, 333
public void setEditable, 812
public void setFont, 786
public void setForeground, 786
public void setLayout, 339

public void setLocation, 786
public void setRows, 812
public void set SelectionBackground, 834
public void setSelectionMode, 834
public void setSize, 786
public void setTabSize, 812
public void setText method, 344, 347, 812
public void SetTitle, 333
public void setVisible, 786
public void setWrapStyleWord, 812
public void showStatus, 788
public void start, 787
public void stop, 787
public void String toString, 802
public void update, 786, 788
public void validate method, 786, 787

Q

question mark (?), conditional operator, **204**

R

radio buttons, 823–828
ragged arrays, **593**
random access memory (RAM), 4–5
reading a single character, 61–63
read statement, 56–61
recursion, 873–897, **874**
  base case, **874**
  decimal to binary conversion (programming example), 889–892
  designing recursive methods, 877
  direct, **876,** 876–877
  general (recursive) case, **874**
  indirect, **876,** 876–877
  infinite, 877
  iteration compared, 888–896
  problem solving using, 878–888
  recursive definitions, 844–877, **874**
  Sierpinski gasket (programming example), 892–896
recursive algorithms, **875**
  designing, 877
recursive case, **874**
recursive methods, **875**
redefining, **643**
reference variables, **115,** 115–118
  as parameters, 414–421
  polymorphic, **665**
  String type, as parameters, 414–418
registering a listener, **349**
relational operators, **180,** 180–183
  arrays, 566–567
  primitive data types, 181–183
repetition, **179,** 249–306
  avoiding bugs by avoiding patches, 295–298
  break and continue statements, 293–295
  choosing a looping structure, 293
  debugging loops, 298–299
  do. . .while looping structure, 288–293
  for looping structure, 278–284
  nested control structures, 299–304
  uses, 250–251
  while looping structure. *See* while loops
reserved words, **30,** 468–469
rethrowing an exception, 749–753
return escape sequence (\r), 70
return statement, 395–398
  syntax, 395
right angle bracket (>), 180
  greater than operator (>), 180
  greater than or equal to operator (>=), 180
rightmost bit, **889**
round method, 386
row order form, **603**
row processing, two-dimensional arrays, **595**
run-time binding, **664**

S

savings accounts, **676**
scope
  classes, 478
  identifiers, **422,** 422–427
searching arrays, 572–574
searching lists, 908–909
  binary searches, **917,** 917–933
  sequential searches, 908–909
secondary storage, **5**
selection, **179,** 179–232, 190–204
  avoiding bugs by avoiding partially understood concepts and techniques, 204–208, 215–223
  comparing floating-point numbers for equality, 202–203
  comparing if. . .else statements with a series of if statements, 200–201
  comparing strings, 223–230
  compound statements, **197**
  conditional operator, **204**
  indentation, 208
  logical operators and logical expressions, 183–185
  multiple, 198–200
  one-way, 190–193
  order of precedence, 185–189
  primitive data types, 181–183
  relational operators, **180,** 180–183
  short-circuit evaluation, 201–202
  two-way, 193–197
selection sort, **909,** 909–913

semantic rules, **29**
semantics, **81**
semicolon (;), statement terminator, **81**
Sen, M. K., 298
sentinel(s), **257**
sentinel-controlled `while` loops, **257,** 257–263
sequences, collating, **34**
sequential access, **178**
sequential searches, **573,** 908–909
`setTime` method, 479–480
shadowing variables, **652**
shallow copying, **477,** 477–478
short-circuit evaluation, 201–202
`showInputDialog` method, 139–140
`showMessageDialog` method, 139, 140–145
Sierpinski gasket, **892**
  programming example, 892–896
signatures, methods, **427**
simple assignment statements, 85–86
single characters, reading, 61–63
single inheritance, **642**
single-line comments, **29,** 29–30
single precision, **35**
single quotation escape sequence (\') 70
`sin` method, 387
software, 6
sorting lists
  insertion sort, 913–917
  selection sort, **909,** 909–913
source code, **73**
source files, **73**
source programs, **10,** 10–11
special symbols, 30
`sqrt` method, 387
standard input stream object, **57**
standard methods, **72**
standard output object, **66**
statements
  action, **190**
  assignment. *See* assignment statements
  blocks of, **197**
  compound, **197**
  declaration, **74**
  executable, **74**
  input, 56–61
  output, **66,** 66–71
  read, 56–63
statements (listed by name)
  `break` statement, 293–295
  `continue` statement, 293–295
  `if` and `if. . .else` statements, 190–204, 215.
    See also selection
  `println` statement, debugging using, 163–165
  `print` statement, debugging using, 163–165
  `return` statement, 395–398
  `static import` statements, **389**
statement terminator (;), **81**
`static` class members, 501–507
`static import` statements, **389**
`static` variables, 503–507
stepwise refinement, **19**
storage, secondary, **5**
string(s), **45,** 45–48
  assignment operator, 229
  character, **45**
  comparing, 223–230
  concatenation, 46–48
  length, **45**
  `new` operator, 230
  null (empty), **45**
  numeric, **138**
`StringBuffer` class, 418–421
`String` class, 121–129, 466–467
  commonly used methods, 122–124
  formatting output using `String` method `format`, 146–149
  object-oriented design, 363–364
`String concat` method, 123
`StringIndexOutOfBoundsException` class, 738, 741
  `InputMismatchException` class, 738
string literal (string constant), **45**
`string` objects, arrays, 574–575
`String replace` method, 123
`String substring` method, 124
`String toLowerCase` method, 124
`String toUpperCase` method, 124
`String` variables, as parameters, 414–418
strongly typed languages, 54
structured design, **19**
structured programming, **19**
stubs, methods, **441,** 441–442
student grade (programming example), 160–162
subclasses, **335, 640**
  constructors, 648–657
  inheritance. *See* inheritance
  using superclass methods in, 642–648
superclasses, **335, 640**
  constructors, 648–657
  inheritance. *See* inheritance
  using methods in a subclass, 642–648
`switch` structures, 208–215
  choosing between `if. . .else` statements and, 215
syntax, 80–84
  actual parameter lists, 394–395, 408
  errors, debugging, 77–80
  formal parameter lists, 394
  method calls, 407
  `return` statement, 395
  value-returning methods, 393–394
  void methods, 407
syntax rules, **29**
system programs, **6**

T

tab escape sequence (\t), 70
Tabulating Machine Company, 3
tail recursive methods, **877**
`tan` method, 387
TBs (terabytes), 7
temperature conversion (programming example), 355–362
terabytes (TBs), 7
ternary operators, **204**
text boxes, 811–816
text fields, **140,** 811–816
text processing (programming example), 609–615
`this` reference, 512–517
throwing an exception, 749–753
`throws` clause, **153**
`toLowerCase` method, 388
top-down design, **19**
`toString` method, 494–500
`toUpperCase` method, 388
Tower of Hanoi problem, 885–888
truth tables, **183,** 183–185
`try` blocks, 728–733
  using in a program, 730–733
two-dimensional arrays, 589–603, **590**
  accessing array elements, 591–594
  initialization during declaration, 594–595
  `length` instance variable, 592–594
  passing as parameters to methods, 599–603
  processing, 595–599
  special cases, 593–594
  storing in row order form, **603**
two-way selections, 193–197
type conversion (type casting), **43,** 43–48

U

UML (Unified Modeling Language), **472**
UML class diagrams, **472,** 472–473
unary operators, **36**
unchecked exceptions, **742**
Unicode, **8**
  collating sequence, 181–183
Unified Modeling Language (UML), **472**
  UML class diagrams, **472,** 472–473
UNIVAC (Universal Automatic Computer), 3
user-defined methods, **384,** 391–404
  value-returning, **391,** 391–398
  void, **392,** 407–411
user interfaces, **330.** *See also* graphical user interfaces (GUIs)

V

`valueOf`, 740
value-returning methods, **391,** 391–398
  syntax, 393–394
variable(s), **50**
  allocating memory with named constants and variables, 48–51
  declaring, 55–56, 473–474
  flag, **263**
  initializing, **51,** 55–56, 60–63
  instance, **469,** 469–470
  names, 50
  primitive data type, as parameters, 411–414
  putting data into variables, 51–55
  reference. *See* reference variables
  shadowing, **652**
variable length parameter list, arrays, 581–589
variables
  `static`, 503–507
  `String`, as parameters, 414–418
`Vector` class, 616–621
  foreach loops, 620–621
  primitive data types, 620
visibility, identifiers, 422–427
void methods, **392,** 407–411
  syntax, 407

W

Web browsers, 13
`while` loops, 251–272
  counter-controlled, **255,** 255–257
  designing, 254
  EOF-controlled, **266,** 266–271
  flag-controlled, **263,** 263–266
  sentinel-controlled, **257,** 257–263
windows, creating, 332–338
World Wide Web (WWW), 13
Wozniak, Stephen, 3
wrapper classes, **139, 370,** 370–377
  as parameters, 421
writing output to a file, 152–153
WWW (World Wide Web), 13

Z

zettabytes (ZBs), 7